The Thesis Committee for Aditya Srikanth

Certifies that this is the approved version of the following thesis:

Characterization and Optimization of JavaScript Programs for Mobile Systems

APPROVED BY

SUPERVISING COMMITTEE:

Vijay Janapa Reddi, Supervisor

Lizy Kurian John

Characterization and Optimization of JavaScript Programs for Mobile Systems

by

Aditya Srikanth, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2013

Dedicated to my loving family.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Vijay Janapa Reddi for his constant guidance and support without which this research would not have been possible. The last two years have been a great learning experience and I would like to thank Dr. Vijay Janapa Reddi for providing a great research atmosphere. I would also like to thank Dr. Lizy Kurian John for her valuable feedback and suggestions during the course of this project. Last but not the least, I am eternally grateful to the love and support of my family.

Characterization and Optimization of JavaScript Programs for Mobile Systems

Aditya Srikanth, M.S.E
The University of Texas at Austin, 2013

Supervisor: Vijay Janapa Reddi

JavaScript has permeated into every aspect of the web experience in today's world, making it highly crucial to process it as quickly as possible. With the proliferation of HTML5 and its associated mobile web applications, the world is slowly but surely moving into an age where majority of the webpages will involve complex computations and manipulations within the JavaScript engine. Recent techniques like Just-in-Time (JIT) compilation have become commonplace in popular browsers like Chrome and Firefox, and there is an ongoing effort to further optimize them in the context of mobile systems.

In order to fully take advantage of JavaScript-heavy webpages, it is important to first characterize the interaction of these webpages (both existing pages and modern HTML5 pages) with the different components of the JavaScript engine, viz. the interpreter, the method JIT, the optimizing compiler and the garbage collector. In this thesis, the aforementioned characterization work was leveraged to identify the limits of JavaScript optimizations.

Subsequently, a particular optimization, i.e. Register Allocation heuristics was explored in detail on different types of JavaScript programs. This was primarily because the majority of the time (an average of 52.81%) spent in the optimizing compiler is for the register allocation stage alone. By varying the heuristics for register assignment, interval priority and spill selection, a clear idea is obtained about how it impacts certain types of programs more than others. This thesis also gives a preliminary insight into JavaScript applications and benchmarks, showing that these applications tend to be register-intensive, with large live intervals and sparse uses, and sensitive to array and string manipulations. A statically-selected optimal register allocation scheme outperforms the default register allocation scheme resulting in 9.1% performance improvement and 11.23% reduction in execution time on a representative mobile system.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

From a humble beginning in 1995, JavaScript has slowly but surely taken over the entire internet. It is estimated that more than 99% of all the websites use JavaScript[3]. The rising popularity is mainly attributed to the ease of development, deployment and portability that is afforded by the high-level of abstraction of JavaScript. In addition, the proliferation of HTML5 and its associated Web 2.0 applications rely heavily on JavaScript for the newly implemented features. When we also consider the fact that over the last 5 years, more than a billion smartphones have been sold around the world and that a vast majority of them support JavaScript[30], it is very clear that JavaScript performance optimization is crucial.

Over the last few years, a lot of effort has been channeled towards improving the performance of JavaScript in modern browsers. Both desktop and mobile implementations of browsers such as Firefox, Chrome and Safari have fully capable and powerful JavaScript engines. Early attempts at processing JavaScript in browsers involved direct interpretation and involved extremely large overheads to the tune of 50x over the corresponding C/C++ implementations[22]. Newer techniques like Just-in-Time compilation

and adaptive optimizations have been incorporated into the current implementations of major JavaScript engines, which have significantly reduced the JavaScript processing times.

In order to understand the available scope for various types of optimizations, it is paramount that we clearly understand the nature of the JavaScript applications themselves. There has been prior work[31] that performs characterization of a wide range of JavaScript benchmark suites to collect both microarchitecture dependent statistics (such as IPC, branch misprediction rate, cache miss rate etc.) as well as microarchitecture independent statistics (such as ILP, control-flow predictability, instruction/data locality etc.). Though this information is very useful in understanding the nature of the benchmarks, it does not reflect on real world websites and upcoming HTML5 websites. To some extent, [27] helps in distinguishing between the JavaScript execution behavior in real world websites & benchmark suites and clarifying the fact that benchmarks are mostly compute-driven while websites tend to be event-driven. However, [27] does not provide any insight on how the characterization data can actually be used to optimize the JavaScript engine.

Towards this purpose, this thesis will study the interaction of JavaScript programs with the different components of the JavaScript engine, viz. the interpreter, the method JIT, the optimizing compiler and the garbage collector. This will help identify potential bottlenecks and opportunities for fine-tuning that have not been identified before in prior work. Leveraging the comprehensive characterization data, this thesis will focus on a particular optimization,

i.e. Register Allocation heuristics at the backend optimizing compiler in order to extract performance improvements.

The remainder of the thesis is organized as follows: Chapter 2 presents the necessary background for JavaScript, HTML5 and the structure of the JavaScript engine. This chapter also focuses on the rapid growth of mobile systems and their impact on the current JavaScript and HTML5 landscapes. Chapter 3 covers the profiling and instrumentation code that was implemented in SpiderMonkey[9], the JavaScript engine present in Firefox. This chapter also contains details about the Pandaboard[24], which was the mobile platform that was used to evaluate all the characterization and optimization work presented in this thesis. Chapter 4 describes the characterization results, along with the valuable insights that were gleaned from this particular detailed study of the various components of the JavaScript engine. Chapter 5 then proceeds to cover the various register allocation heuristics along with the results of performing this optimization. Finally, Chapter 6 concludes by summarizing the work accomplished in this thesis and setting the stage for future work in this fascinating field.

Chapter 2

Evolution of the JavaScript Landscape

JavaScript has its roots traced back to 1995 where it was developed at Netscape as a simple and powerful scripting language in order to provide more accessibility to non-Java programmers and web developers to create applications similar to native Java applets. Today, almost two decades since it was introduced, JavaScript has burgeoned into a formidable force that is present in 99% of all websites[3], and has garnered the support of thousands of web developers all over the world. Modern day frameworks like node.js[19] and Meteor[16] provide both server-side and client-side JavaScript programmability enabling developers to create rich, immersive web applications in a fraction of development time compared to older techniques. To delve deeper and study this change, this chapter elucidates the evolution of JavaScript, along with the associated challenges involved that make JavaScript compilation challenging. The chapter then proceeds to cover the evolution of JavaScript engines in browsers, and concludes with an overview of the current scenario with the widespread proliferation of HTML5 and fully featured JavaScript support on mobile devices.

2.1 A Brief History of JavaScript

JavaScript is a prototype-based scripting language characterized by its dynamic and weakly-typed nature. It was initially christened LiveScript to showcase its dynamic nature, but was quickly renamed to JavaScript. It was aimed at an audience that consisted of mostly web designers and developers who needed to be able to tie into page elements (such as forms, or frames, or images) without a bytecode compiler or knowledge of object-oriented software design. JavaScript aimed to bring full interactivity and sophisticated user interface and typography concepts to the formerly static web. It could simply be inserted directly into existing HTML pages and was tightly integrated into the browser and could easily react to user events.

Early implementations of JavaScript in Netscape Navigator[5] did not have good support in the form of integrated development environments or debuggers. It also had a fair share of security violations where users could be tricked into running some malicious scripts on the browser. Subsequent releases of Navigator included support for script-driven interaction with plugins, along with a more rigorous security model. Even in the early days, JavaScript was capable of being used as a server-side language to query databases in Netscapes LiveWire[15].

JScript was Microsoft's version of JavaScript which was part of the Internet Explorer 3.x but it was not entirely compatible with the original JavaScript implementation leading to inconsistencies in website development. This led to the standardization of the language by the European Computer

Manufacturers Association (ECMA) called ECMAScript which was launched in 1998. This time period also coincided with the infamous browser wars leading to many more browser-specific implementations which eventually paved way to the Document Object Model (DOM) specification[32] governed by the W3C. This is currently the form of JavaScript (version 1.8.5) that is prevalent today with clearly defined context-specific object models. Programmers have access to a wide variety of DOM Methods and Event handlers, which are available as JavaScript objects that they use to successfully manipulate the HTML elements in a web page.

Initially, the naming convention lead to multiple misunderstandings between the Java and JavaScript. The main difference between Java and JavaScript is that Java applications are standalone while JavaScript programs must (primarily) be placed inside an HTML document for correct functionality. Feature-wise, the Java programming language is much larger and more extensive than JavaScript. This requires that Java programs be compiled into bytecode before it can be run on the system via a Java Virtual Machine (JVM). On the other hand, JavaScript can be executed from plain-text source code directly on the browser itself. This also highlights the fact that any changes to the Java application requires recompilation into the updated bytecode while changing JavaScript programs just involves direct modification to the source. Java programming is very rigid as it is a strongly-typed language and requires every object to be explicitly spelled out. On the contrary, JavaScript is more forgiving to the programmers in the sense that it is dynamically typed and

allows more freedom in creation and manipulation of objects. This fact also leads to multiple compilation challenges that are highlighted below.

The main compilation challenges for JavaScript can be classified into two categories: Prototype-based rather than class-based, and being dynamically typed rather than statically typed. The former refers to the fact that in a prototype-based language, there are no classes present and the only way to reuse (or inherit) behavior is by cloning the existing objects which serve as 'prototypes'. The following source code listing is an example of prototypal inheritance.

```
var foo = {one: 1, two: 2};
var bar = Object.create( foo );  //Prototypal inheritance
bar.three = 3;
bar.one;     // 1, inherited from foo
bar.two;     // 2, inherited from foo
bar.three;   // 3, specific to bar
```

In a class-based language, every single instance of a particular class will have the same behavior: the same set of fields to store data and the same set of methods to operate on the data. A compiler would need to generate optimized code for the class just once and it can be reused for all the instances of that class. Being a prototype-based language, even for a simple operation such as looking up a particular field in the object, the JavaScript interpreter has to essentially look inside a dictionary mapping from field names to field values which is a slow and expensive operation. As exemplified in the source code above, there is no requirement for object `bar` to have just one more new member called `three`, but it could have multiple new members. Thus, a fixed

layout is not possible in this case. This is in stark contrast to the fact that in a traditional compiled language, the compiler has the knowledge of the exact layout of each field in a particular class and can directly access any field of any object of the class. In addition, the extra effort of generating common reusable code in JavaScript far outweighs the performance improvement provided by the code generation.

The latter issue regarding the fact that JavaScript being weakly-typed is relatively straightforward. In a statically typed language, the every variable or field has a singular, well-defined type and that the variable can store only objects of that particular type. The compiler can easily detect mismatched types before a particular operation and give a corresponding error message. However, in JavaScript, as with other dynamically typed languages, variables and fields can store values of any type thereby pushing the evaluation of these statements until runtime when the type can be determined with certainty.

```
var x = 5;     // x is an integer
var y = "37";  // y is a string
var z = x + y; // z is (?)
```

In the above source code example, the dynamic typed nature of JavaScript is showcased. In the above example, JavaScript would convert the integer 5 to the string "5", perform string concatenation, and produce the string "537" as the value of `z`. Errors are thrown by the interpreter only if it does not know how to handle the particular types at runtime. There are tradeoffs associated with both these techniques - detecting errors early in the compilation phase rather than at runtime versus having more freedom to code powerful features

like interfaces, templates etc. without having to worry about the complexity of their implementation. Additional compiler techniques like type-inference, type-specialization or multi-versioning have been developed in the context of JavaScript to overcome this difficulty. These will be covered in detail in the following section which introduces the JavaScript Engine.

2.2 JavaScript Engines

This section will first cover the overall structure of the browser before diving into the details of the JavaScript Engine. Subsequently, the evolution of JavaScript engines is also highlighted showcasing their increased capabilities in recent years.

2.2.1 Structure of the Web Browser

JavaScript had always been designed to be used in conjunction with the browser, and as expected, a major portion of the browser implementation consists of the JavaScript engine. This section focuses on the overall structure of the browser, and briefly describes each of the constituents. Figure 2.1 showcases the block diagram of a typical browser.

The user interface block primarily consists of the on-screen navigation menus which is visible to the end-user. The browser engine is an interface that facilitates the interaction between the user and the underlying rendering engine. The rendering engine is responsible for the various stages like parsing HTML, CSS and applying corresponding styles and finally creating a DOM

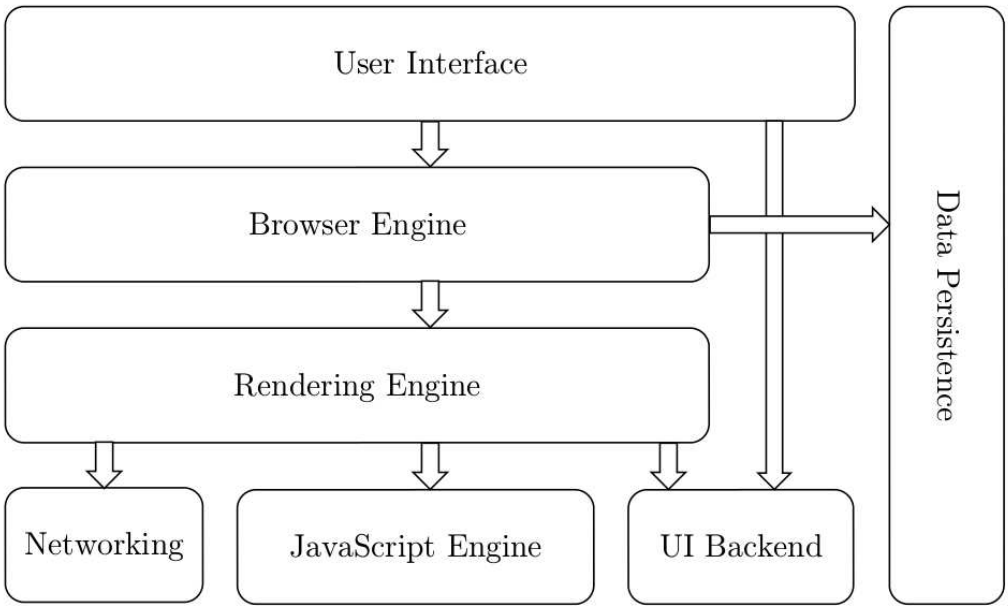

Figure 2.1: Overview of the Browser Architecture

tree which is the main data structure that represents the internal structure of the website. The rendering engine finally constructs the render tree after parsing the DOM tree which is then painted on the screen. The Networking layer is responsible for all networking calls and protocols (such as HTTP, FTP etc.). The UI backend takes care of platform independent graphics like rendering windows, dialog boxes etc. that uses the underlying operating system interfaces. The Data Persistence layer handles local data storage like cookies. In the case of HTML5 applications, which permit local persistence data storage across sessions, managing the complete lightweight local database is also taken care of by this layer. The final constituent is the JavaScript engine itself, which will be the primary focus of the following section.

2.2.2 Structure of the JavaScript Engine

The JavaScript Engine is a specialized virtual-machine environment that parses and executes JavaScript code present in the webpage. The first JavaScript engine that was developed at Netscape was written in C++ and primarily consisted of just the interpreter for JavaScript bytecode that was generated after parsing and constructing the Abstract Syntax Tree (AST). Many other implementations of the basic JavaScript engine followed, such as Rhino which was written entirely in Java[10], Squirrelfish[33] for Safari (on Mac OSX) and Chakra for Internet Explorer (on Windows) to support multiple platforms and multiple browsers. A significant step forward in this field began with the introduction of a Just-in-Time (JIT) compiler in Google's V8 JavaScript Engine[17] in 2008. The JIT compiler generates small fragments of native code and compiles JavaScript programs in a piecemeal fashion, rather than relying on pure interpretation. The compiled code is also optimized (and re-optimized) dynamically at runtime based on the gathered profile of the executed code. Following suit, most of the current JavaScript engines also had their own implementations of a JIT incorporated in them.

Before proceeding further, Figure 2.2 and Figure 2.3 highlight the general structure of the JavaScript engines: older implementations that relied on pure interpretation and newer implementations that incorporate all the parts of a modern virtual machine - Interpreter, JIT compiler, Backend Optimizing compiler, Profiler, Garbage Collector etc.

Figure 2.2 shows the layout of a basic interpreted JavaScript Engine.

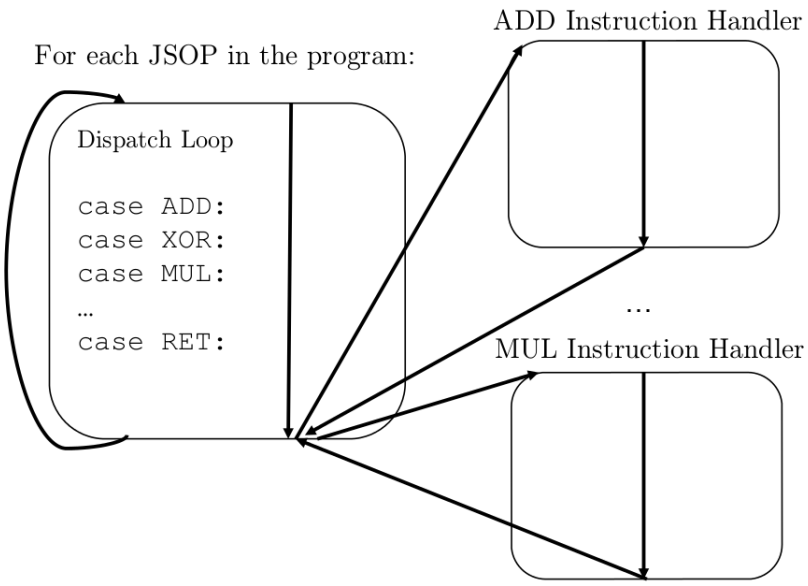

Figure 2.2: Basic Structure of a generic JavaScript Engine

As is typical of such systems, the main portion consists of a single large switch statement with specific handling routines for each of the different JavaScript bytecodes. The runtime has to sequentially read the parsed JavaScript bytecode of a particular program and then jump between different handling routines. Specific optimizations such as direct threading, where control jumps directly from one handling routine to the next instead of jumping back to the central switch statement, was implemented in Mozillas JavaScript engine called SpiderMonkey[9]. The interpreter maintains a JavaScript-to-JavaScript call stack as well as a JavaScript-to-C-to-JavaScript call stack. This is mainly because a JS-to-JS function call pushes a JavaScript stack frame without growing the C stack frame, thus requiring two independent stacks. But since JS-to-

C-to-JS call stacks are common, the interpreter was designed to be reentrant. Some of the JavaScript bytecode operations have multiple cases, depending on the type of their arguments. Another kind of optimization that arose from this was the inlining of commonly occurring cases via macro expansion. This essentially meant that after the compiler had finished its preprocessing stages, the common cases were inlined and resulted in a larger scope for optimizing the code. However, after the successful release of the Google V8 engine, the newer versions of SpiderMonkey were forced to include their own implementation of JIT compilers and generational garbage collectors[23].

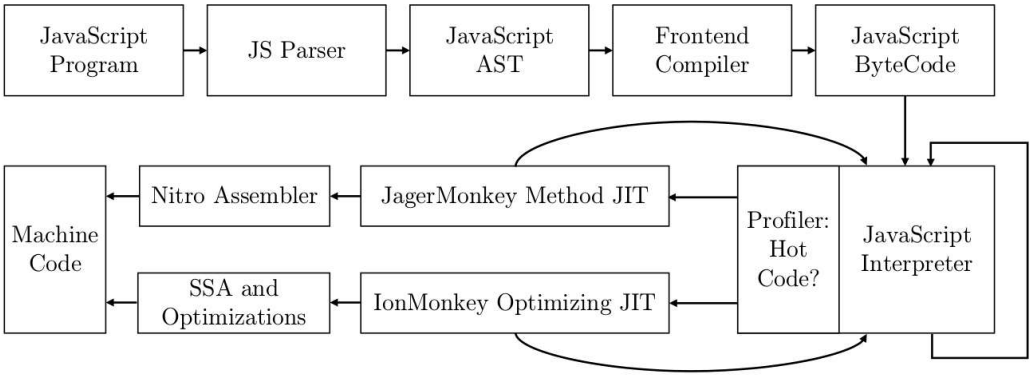

Figure 2.3: Detailed Layout of the SpiderMonkey JavaScript Engine

Figure 2.3 illustrates the layout of a modern JavaScript Engine, in this case Spidermonkey, with its entire set of internal constituents. The general flow of control is as follows: The frontend compiler initially parses the JavaScript source code and produces a "script" object that contains the JavaScript bytecode, source annotations and a pool of string, numeric and identifier literals. In addition, the "script" object also contains other objects and functions defined

in the source code, which may themselves contain their own, nested scripts. The frontend compiler consists of a recursive-descent parser that produces an AST, a tree-walking bytecode generator which also performs some basic optimizations such like Constant Folding, and a decompiler which can reconstruct the JavaScript source code by translating postfix bytecode into infix source using the annotated information. The generated bytecode is then passed on the interpreter which starts processing them sequentially. The interpreter structure is the same as the one described in the preceding paragraph. The profiler is responsible for collecting the execution profile of a piece of code. If a particular function is hot or consists of hot loops, then it is beneficial to avoid the interpretation overhead each time and the JIT compiler is invoked.

The first JIT that was written for the SpiderMonkey JavaScript engine was called TraceMonkey[14], which compiles hot traces into machine code. TraceMonkey records the relevant control flow and data types before deciding on the hot trace to convert to machine code. Subsequently, SpiderMonkey implemented a well-defined Type Inference (TI) engine along with their next generation JIT called JägerMonkey. This rendered the implementation of TraceMonkey obsolete which was discontinued from Firefox 11.

JägerMonkey is a whole-method JIT that converts the bytecode of entire functions to machine code for faster execution. This was able to provide performance improvements in cases where TraceMonkey could not generate stable machine code. While typical compilers work by constructing and optimizing a control flow graph representing the function, JägerMonkey instead

operates by iterating linearly forward through bytecode. The compiler optimizations were sacrificed for getting the maximum speed possible out of the system, considering that JavaScript variables change their types often. Another critical optimization in JägerMonkey was the use of polymorphic inline caches (PICs)[21] which perform faster object type lookups.

The latest implementation of the JIT for SpiderMonkey is called IonMonkey [11] which further aims to improve upon JägerMonkey and enable many new optimizations that were not possible in the previous architecture. IonMonkey was designed as a more traditional compiler: the bytecode is converted into a control flow graph using static single assignment (SSA) [6] for the intermediate representation. The adoption of SSA enables a plethora of traditional compiler optimizations that were simply not possible in the earlier implementation.

Currently, IonMonkey implements the following optimization passes: Loop-Invariant Code Motion (LICM) which involves moving certain loop-independent instructions outside the loop whenever possible; Global Value Numbering (GVN) which is a powerful reuse optimization for removing redundant code; Linear Scan Register Allocation (LSRA) which does an efficient register allocation in linear time as compared to Graph-coloring register allocation; and Dead Code Elimination (DCE) which focuses on removing unused instructions. This clearly showcases the wide scope of optimization potential of IonMonkey and is also the focus of the second half of this thesis.

The last component of note in the JavaScript engine is the garbage

collector. When a JavaScript program creates lots of temporary objects and runs out of space to allocate new objects, the garbage collector is invoked to clear out the dead objects and make space in the heap to allocate new objects. The particular implementation in SpiderMonkey is a mark-and-sweep, non-exact collector. The marking phase is done in different slices so as to not affect the running of the JavaScript program, and the collection is relegated to a background thread again to minimize the latency to the end-user. In addition, it is possible to customize the frequency of invocation of the garbage collector to suit the needs of different types of JavaScript applications.

This section covered all the major components of a modern JavaScript engine and how they interact with each other. Though these engines typically have good performance for processing current day websites, it is predicted that complexity of websites is bound to increase especially with the adoption of the new HTML5 standard. The following section gives a brief overview about HTML5 and how it is expected to change the web browsing scenario.

2.3 The Rise of HTML5

One of the biggest game-changing developments over the last couple of years has been the introduction and steady growth of HTML5. It is essentially the fifth revision of the original HTML standard, and at its core aims to improve the support for a variety of multimedia features such as graphics, audio, video etc. It subsumes all existing standards including HTML 4, XHTML 1 (eXtended HTML) as well as DOM based HTML (tags such as

`GetElementByID`). This wide scope gives it immense potential to replace or destabilize existing environments and is bound to have an immense impact in the current scenario. This section first covers the new tags specific to the HTML5 standard, followed by its impact on the desktop and mobile landscape.

The newly included tags in the HTML5 standard are diverse in nature, but the most challenging ones focus mainly around multimedia, viz. `video`, `audio` and `canvas` tags. Most of the new tags are tightly integrated with specific JavaScript APIs that enable their functionality. Consider the simple example of the `canvas` tag. It just initializes an empty canvas on the browser screen, but the associated drawing methods to render lines or arcs rely completely on JavaScript. The more complex the graphics that is rendered on the canvas, the more stress on the JavaScript engine to process it. Other commonly used APIs in HTML5 applications are the features to drag-and-drop items, geolocation and location tracking, locally stored application data, real-time communication protocols etc. These listed features just scratch the surface of what HTML5 is truly capable of, but present a good case to motivate the use-cases of future applications. Figure 2.4 showcases the constituents of the current HTML5 standard.

From the context of mobile systems, HTML5 is poised to destabilize the App dominated environment. Developers can now develop fully functional HTML5 web applications rather than tailor their applications to suit different devices (of varied compute and display capabilities) and different operating systems. This can potentially save hundreds of man hours in the App develop-

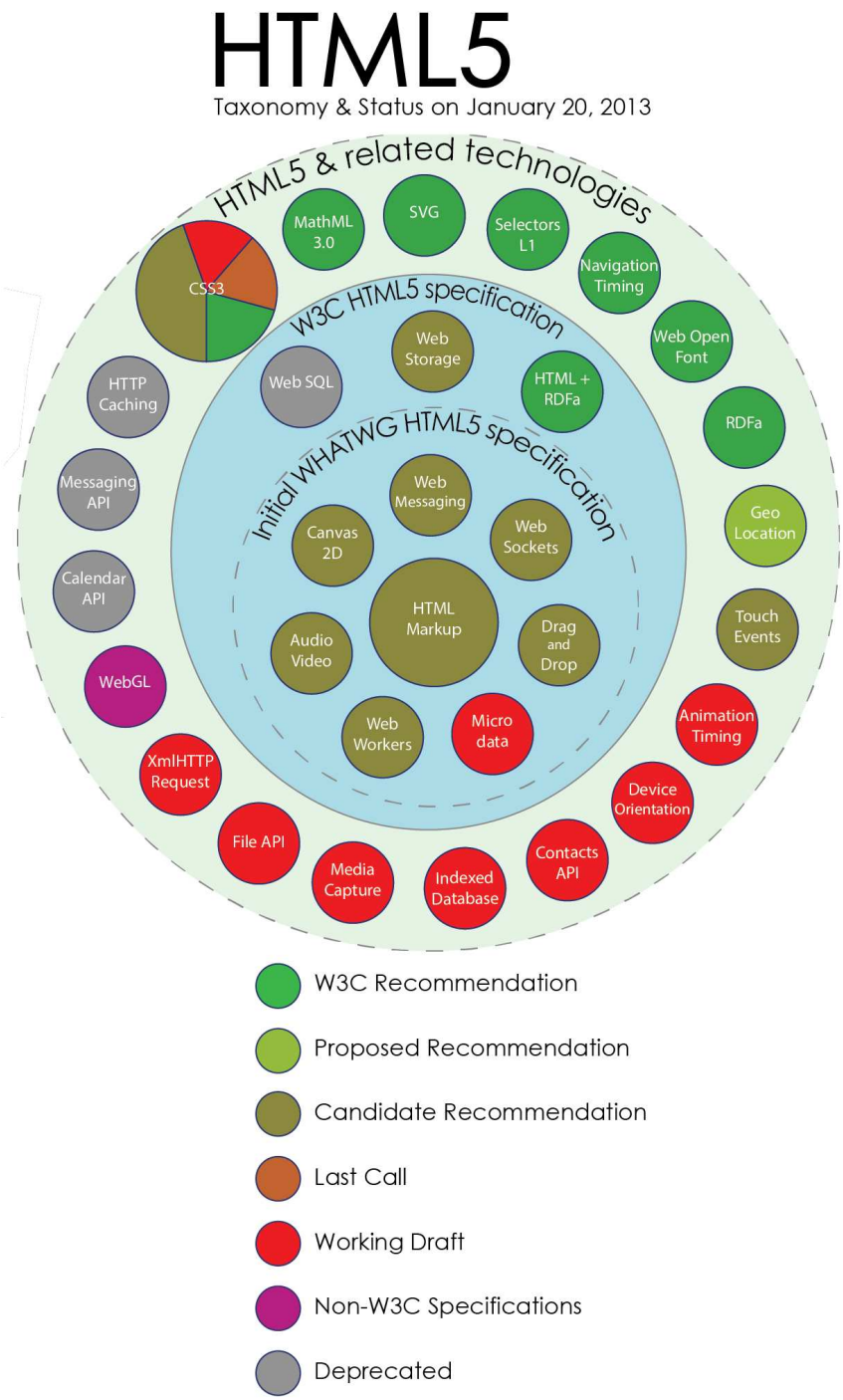

Figure 2.4: The HTML5 standard

ment cycle. However, there is a trade-off that natively developed applications are usually faster than their corresponding HTML5 applications. This again is a crucial point in the context of this research. Continuing advancements in optimizing the JavaScript engine is bound to reduce the performance gap between native applications and HTML5 web applications. A specific example that can be used to justify the above statement is the introduction of OdinMonkey in the latest nightly build of Firefox. This is a very specific and targeted optimization on asm.js[2] which is a low-level, efficient target language for compilers. The performance boost provided by this subset of JavaScript is so significant that it performs within a factor of 2x of the native C++ implementation. This is especially beneficial while designing browser based games that do not need any external plugins, as demonstrated by the Unreal Engine 3 which consistently managed a highly responsive framerate[13].

The preceding paragraphs gave a clear indication about the conception and growth of HTML5. Previous surveys have shown that 34% of the Top 100 websites as recorded on Alexa.com currently use HTML5 specific tags[29]. Newer websites that are completely built with HTML5 are bound to be more demanding. In order to verify this claim, an experiment was performed using the Gecko Profiler add-on installed on Firefox 17. The Gecko Profiler can be used for a wide variety of statistics, but this experiment focused purely on profiling the JavaScript content, and how much time is spent in the JavaScript engine while loading the page. The sample consisted of the Top 10 Hottest websites as listed on Alexa.com and the Top 10 HTML5 websites as of 2012[7].

Table 2.1: Percentage of time spent in the JS engine for regular websites

Top 10 Hottest Websites	**% time spent in JS Engine**
Google	16
Facebook	19
YouTube	4.8
Yahoo!	9.9
Baidu	1.8
Wikipedia	0.8
Windows Live	39.1
QQ.COM	15.7
Amazon.com	19.9
Twitter.com	1.6
Arithmetic Mean	**12.7**

Table 2.2: Percentage of time spent in the JS engine for HTML5 websites

Top 10 HTML5 Websites	**% time spent in JS Engine**
The Wilderness Downtown	62.8
Three Dreams of Black	69.5
360 Zurich	58
Soul Reaper	56.2
Universaries	48
The Expressive Web	73.4
CNN Ecosphere	52.2
Art of Stars	64.4
This Shell	76.1
Lost Worlds Fair	42
Arithmetic Mean	**60.26**

On an average, the Top 10 Hottest websites spend around 12.7% in the JavaScript engine, while the Top 10 HTML5 websites spend a staggering 60.26% in the JavaScript engine. The individual details of each of the websites are shown in Table 2.1 and Table 2.2. This initial experiment clearly demonstrates the need for optimizing the JavaScript Engine in order to effectively process up-and-coming HTML5 websites.

In conclusion, this chapter provided a brief history of JavaScript, followed by the evolution of the JavaScript engines with a specific focus on SpiderMonkey, the implementation in Firefox. Subsequently, this chapter focused on the ascent of HTML5 based web applications, and provided the motivation for optimizing the performance of the JavaScript Engine.

Chapter 3

Instrumentation Details and Experimental Setup

This chapter focuses primarily on the instrumentation work in the source code of the SpiderMonkey JavaScript engine to collect the required statistics. Source-level instrumentation was required in this research to allow finer control of instrumentation points and access to the internal data structures maintained by SpiderMonkey.

The second half of this chapter focuses on the platform that was used for testing the modified SpiderMonkey engine. For this purpose, a Pandaboard[24] was used because it was a representative smartphone platform with reasonably good Linux support. A brief description of the benchmark suites, viz. SunSpider[34] and v8[18] that were used for the evaluation also follows.

3.1 SpiderMonkey Instrumentation Code

The internal constituents of SpiderMonkey were covered in considerable detail in the previous chapter. This section will focus specifically on the various instrumentation points along with their intended functionality.

The entry point from the browser into the JavaScript engine is present

in the `jsapi.h` and `jsapi.cpp` files. These files contains the public API that is used by almost all client code. Timing functions were inserted to compute the time spent in different functions (such as time spent in Interpreter, time spent in JägerMonkey etc.) in order to identify which functions were the hottest. Individual timestamps of function entry and function exit were also maintained in order to get a time-sensitive trace.

The core of the instrumentation was implemented inside `jsinterp.cpp` which has the bytecode interpreter. It is worthwhile to note that most of the implicit state of each interpreter instance is stored in a single object of the JSContext class which is passed as a parameter into almost all functions in SpiderMonkey. Control first enters the interpreter before decisions are made whether to jump into JägerMonkey or IonMonkey. Entry points into the respective JIT compilers are called `JaegerShot()` and `IonCannon()` respectively. Additionally, there are well-defined compiler flags that enable or disable the JIT compilers altogether. These were used extensively for the limits of JavaScript optimization study described in Section 4.2.

Additional modifications to the source code were done inside the implementations of both the JITs. For JägerMonkey, modifications were done in `Compiler.cpp` and `MethodJIT.cpp` to collect the statistics about when JägerMonkey is in compilation stage and when it is actually executing the native code. The function of interest were `jm_compile()` and `jm_normal()` respectively. Similarly, in the case of IonMonkey, modifications were done inside `Ion.cpp` to distinguish between the time spent on the compilation overhead

Table 3.1: Instrumented Functions and their Description

Function Name	**Description**
`interpret()`	Call to start the interpreter
`jm_compile()`	Compilation phase of JägerMonkey
`jm_normal()`	Execution phase of JägerMonkey
`ion_compile()`	Compilation phase of IonMonkey
`ion_cannon()`	Execution phase of IonMonkey
`js_gc()`	Call to start the garbage collector
`execute_script()`	Entry point of browser code into JS engine

(which involves constructing the Control Flow Graph, translating to SSA, and performing the optimization passes) and the time spent in actually executing the optimized code. The functions of interest in this case were `ion_compile()` and `ion_cannon()` respectively. Finally, the rest of the instrumentation was done in the garbage collector. Calls to the garbage collector are present both from `jsinterp.cpp` and from `jsapi.cpp` for clean-up after a particular script object has been executed to completion. Both these places were instrumented for calls to the garbage collector. The function of interest in this case was `js_gc()`. It is also interesting to note that the garbage collector has compiler knobs to control the frequency with which the garbage collector is called. This is referred to as different 'zeal' levels in the SpiderMonkey parlance, and can be set to invoke the collector either after a fixed number of allocations on the heap, or after potentially dangerous allocations, or when the heap is full etc. This plays a crucial role in the performance of the JavaScript engine as a whole.

Table 3.1 contains a summary of all the functions that were instrumented along with their respective purposes.

3.2 Experimental Setup - Pandaboard

The PandaBoard is a low-power, low-cost single-board mobile development platform based on the Texas Instruments (TI) OMAP44xx system on chip (SoC). There are currently two versions of the PandaBoard available in the market, viz. the PandaBoard and the PandaBoard ES. The former is based on the OMAP4430 SoC while the latter is based on the OMAP4460 SoC which has the CPU and GPU running at higher clock rates. For this research, the PandaBoard ES was used all subsequent descriptions are applicable to this model only. The PandaBoard ES which has the TI OMAP 4460 SoC features a dual-core ARM Cortex-A9 MPCore with Symmetric Multiprocessing at upto 1.2GHz each and a POWERVR SGX540 GPU[25] along with 1GB of DDR2 SDRAM. It also has a host of other peripherals that make it convenient to use, and acts like a fully featured mobile system prototype. The video output from the PandaBoard ES was via a HDMI out and was connected to a screen of resolution 1080p, which is typical in high-end smartphones today. The rest of the ports are highlighted in Figure 3.1.

The board was chosen because of its easy availability and widespread Linux support. The OS installed on the PandaBoard was the Ubuntu 12.04 distribution with Linux kernel 3.2.0-1409. Then, Firefox 18 source code was downloaded and instrumented as explained in the previous section. Subse-

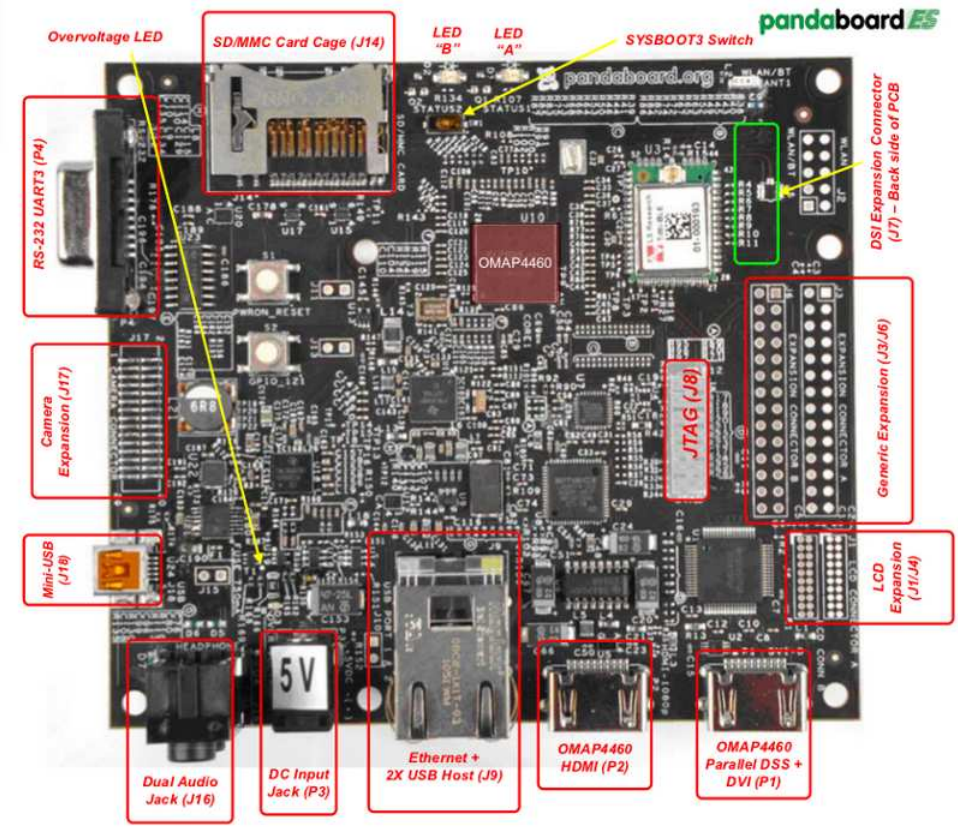

Figure 3.1: Pandaboard Schematic Diagram

quently, the JavaScript Engine was built as a standalone shell for running the benchmarks suites and collecting their profile information. The entire Firefox browser was also built in order to test the performance of real-world websites. Although the results presented in this thesis are based on the SpiderMonkey JavaScript Engine in Firefox, the principles and observations can easily be extended to and applied on any of the other browsers. SpiderMonkey was chosen for this study because of well-defined and separate units of the Interpreter and various JITs. It was also found to be easier to instrument, and was

well-engineered with many command-line and compiler options to tweak its performance.

3.3 Description of the Benchmark Suites

For the purpose of evaluation, the two most popular JavaScript benchmark suites, viz. Sunspider and v8 were used in this research. The following paragraphs give a brief description about the type of benchmarks in each of the suites.

3.3.1 Sunspider Benchmarks

Sunspider is a benchmark suite that aims to measure JavaScript performance on tasks that are relevant to current and near future use of JavaScript, and was initially released in 2007 by the Webkit team. However, Sunspider does not test the DOM and other browser-related APIs. There are multiple benchmarks (a total of 26) in this suite but they can be classified into the following categories: *3d*, *access*, *bitops*, *controlflow*, *crypto*, *date*, *math*, *regexp* and *string*. *3d* mainly focuses on the kind of computations that are prevalent in 3D rendering and stresses the floating point math and array accesses of JavaScript. *access* stresses the JavaScript arrays, object properties and variable accesses. *bitops*, as the name suggests, consists of benchmarks that focus on bitwise operations, which are useful for games, mathematical computations, and encoding/decoding schemes. An interesting point to note here is that bitwise operations are the only math operations in JavaScript that are performed

on integers, while the rest of the math operations are purely floating point. *controlflow* encompasses most control flow constructs such as loops, recursion and conditionals. The *crypto* section consists of real world cryptography code, and stresses both bitwise operations and string operations. The *date* subset of benchmarks attempt to stress the JavaScript "date" objects specifically. *math*, *regexp* and *string* cover a wide variety of mathematical computations, regular expressions and string processing operations respectively. Table 3.2 shows all the SunSpider benchmarks in their respective categories.

3.3.2 v8 Benchmarks

The v8 benchmark suite was released by Google in conjunction with their v8 JavaScript engine. The latest iteration of this suite was released in 2011 and consists of 8 benchmarks. The main differences between the v8 and SunSpider suites is the fact that most of the SunSpider benchmarks have really short runtimes (primarily attributed to the fact that JavaScript processing had become much better in that timeframe) while v8 benchmarks run for significantly longer. The set of benchmarks in the v8 suite along with a brief description of each is present in Table 3.3. *Richards* is a benchmark that mainly focuses on property load/store and function calls. It also tests code optimization and elimination of redundant code to a smaller degree. *DeltaBlue* focuses on polymorphism and object-oriented style programming. *Raytrace* stresses the creation of prototype library objects while *Regexp* stresses regular expressions. *NavierStokes* heavily involves reading/writing double precision arrays

Table 3.2: Sunspider Benchmark Suite

Category	**Benchmarks**	**Pandaboard Runtime(ms)**
`3d`	3d-cube	107.7
	3d-morph	68.5
	3d-raytrace	132.6
`access`	access-binary-trees	13.8
	access-fannkuch	50.8
	access-nbody	46.9
	access-nsieve	22.2
`bitops`	bitops-3bit-bits-in-byte	16.8
	bitops-bits-in-byte	21.8
	bitops-bitwise-and	43.9
	bitops-nsieve-bits	36.7
`controlflow`	controlflow-recursive	17.4
`crypto`	crypto-aes	63.0
	crypto-md5	58.2
	crypto-sha1	31.7
`date`	date-format-tofte	108.2
	date-format-xparb	231.8
`math`	math-cordic	28.8
	math-partial-sums	48.1
	math-spectral-norm	23.5
`regexp`	regexp-dna	80.7
`string`	string-base64	37.4
	string-fasta	71.4
	string-tagcloud	113.9
	string-unpack-code	178.4
	string-validate-input	83.8

Table 3.3: v8 Benchmark Suite

Benchmark	**Description**
Richards	OS kernel simulation benchmark
Deltablue	One-way constraint solver
Raytrace	Ray-tracing benchmark
Regexp	Regular Expression benchmark
NavierStokes	2D Navier Stokes equations solver
Crypto	Encryption and Decryption benchmark
Splay	Splay-tree data manipulation benchmark
EarleyBoyer	Classic Scheme benchmark

and floating point math operations. *Crypto*, as the name implies, focuses on encryption and decryption algorithms and mostly involves bit operations. *Splay* and $Earley - Boyer$ focus on fast object creation and destruction and are meant to exercise the automatic memory management subsystem. The v8 benchmark suite measures the time taken to complete the test and then assigns a score that is inversely proportional to the runtime. It is calculated as the geometric mean of the individual results as compared to a reference system (of score 100). This procedure also ensures that the results are comparable across different systems and platforms as long as the same version of the benchmark suite is used. In simple terms, higher score is better and it implies that the benchmark finished execution much quicker than on a system with lower score.

The following chapter dives into the characterization details and results garnered from the study.

Chapter 4

Characterization of JavaScript Programs

In order to fully understand the scope for JavaScript optimizations, it is crucial to understand the nature of JavaScript programs and how they interact with the JavaScript engine. This chapter focuses entirely on the characterization of JavaScript programs.

4.1 Prior work in JavaScript Characterization

Progressing hand-in-hand with the development of JavaScript engines was the relevant characterization work to understand the nature of JavaScript programs. One of the earliest and most interesting works in this regard was the JSMeter project by Ratanaworabhan et al [27] who did a thorough examination of the characteristics of real world webpages as compared to the existing benchmark suites. This study measured characteristics such as hot functions, allocated heap size, number of event handlers etc. for both websites and benchmarks and concluded that benchmarks are not at all representative of real websites. They concluded that benchmarks are mainly compute intensive and batch-oriented while real world websites are mostly event driven.

Furthermore, Tiwari et al [31] performed an in-depth architectural char-

acterization of both the v8 and SunSpider benchmarks suites. They evaluated both microarchitecture dependent and microarchitecture independent statistics. The main result of their study was that the v8 benchmarks have a higher instruction level parallelism, higher branch frequency but good branch prediction accuracy as compared to SunSpider which have a lower branch frequency but higher branch misprediction rates. They also conclude that there is significant redundancy in the SunSpider benchmarks which exhibit significant similarity for a wide range of execution statistics, but it is not the case in the v8 benchmark suite. They also evaluated the architectural characteristics such as cache access rates, TLB misses etc. in both the SunSpider and v8 suites.

Another interesting study was performed by Fortuna et al [8] who studied data dependences and control dependences in JavaScript programs to study the limit of parallelism possible. Their results indicate that though parallelized JavaScript programs show an average of 8.9x speedup over sequential execution, they do not have significant levels of loop-level parallelism as is common in scientific computations. In this case, the difference between benchmarks and real webpages wasn't as drastic as compared to [31] and [27].

Though these previous works have characterized JavaScript programs along different dimensions, they have not considered the interaction of these JavaScript programs with the different constituents of the JavaScript engine. This is the main focus of the characterization work done in this thesis. By tracing the execution of the benchmarks across the different components of the SpiderMonkey JavaScript engine, i.e. the interpreter, JägerMonkey, Ion-

Monkey, and the Garbage collector, a lot can be inferred about which section of the JavaScript engine is worth optimizing for different use-cases. This is explained in detail in the following section.

4.2 Opportunistic Evaluation of the Optimization Space

The opportunistic evaluation of the optimization space mainly consists of two parts. First, the impact of using various components of the JavaScript engine is identified independently. This experiment drives home the importance of IonMonkey and the benefits of including it for JavaScript optimizations. The second part includes a more comprehensive study of the time spent in each of the sections, along with the characterization of both the SunSpider and v8 Benchmark suites.

4.2.1 Impact of Individual Components

In the previous chapter, the instrumentation changes to the source code of SpiderMonkey was explained in detail. In addition to the source-level changes, the presence of compiler flags to enable/disable entire sections of the JavaScript engine presented a good starting point for the opportunistic analysis of the optimization space. The main components that were tested initially included the Interpreter, JägerMonkey, IonMonkey and the Type Inference engine. Proceeding in the order of evolution of the SpiderMonkey JavaScript engine, a set of experiments were conducted that tested each of the components individually and in conjunction with each other. The details of these

experiments in presented in the following paragraph. The graphs for these experiments showing the scores and execution times are presented in Figure 4.1 and Figure 4.2 respectively.

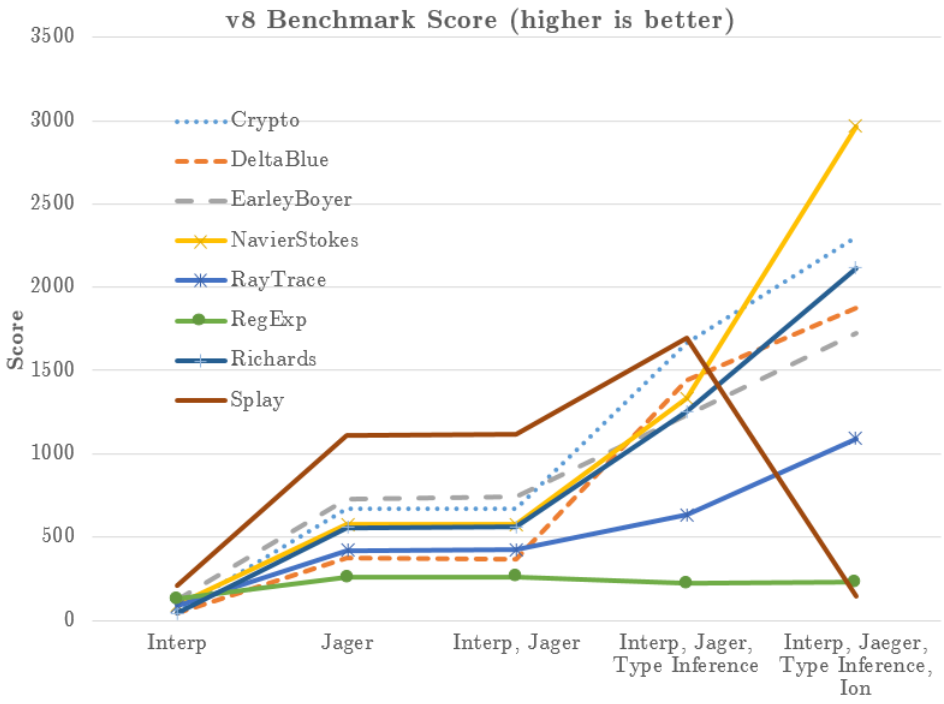

Figure 4.1: Opportunistic Evaluation - v8 Benchmark Scores

Initially, the v8 benchmark suite was run using the interpreter alone. This was representative of the time when there were no JIT engines available to process JavaScript. This was achieved by running the benchmarks using the `--no-jm --no-ti --no-ion` flags which disabled JägerMonkey, the Type Inference engine and IonMonkey respectively. As expected, the scores obtained by this experiment were the lowest and the execution times were the highest. Next, the performance using JägerMonkey alone was tested using

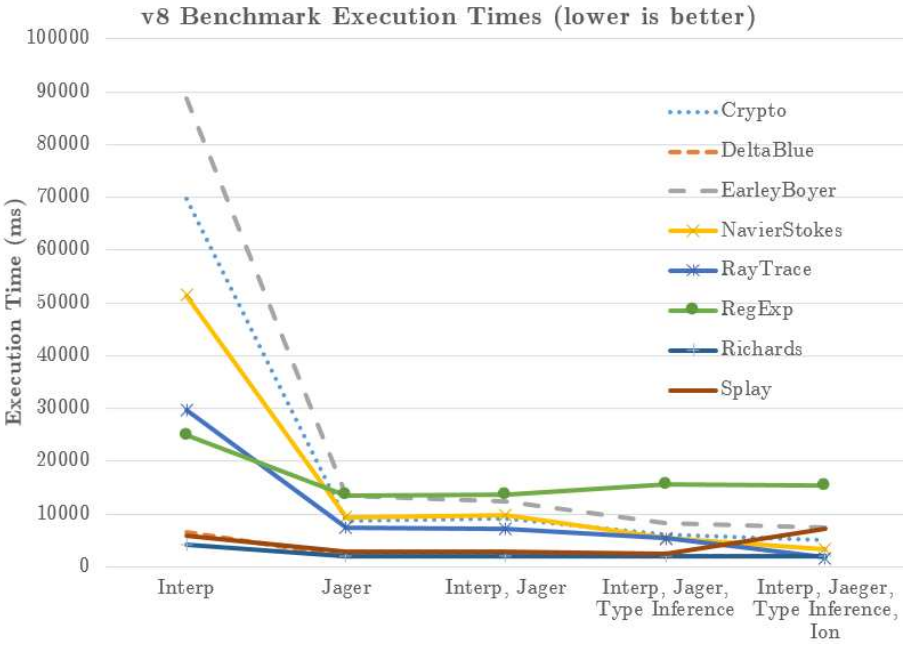

Figure 4.2: Opportunistic Evaluation - v8 Benchmark Execution Times

the `-a --no-ti --no-ion`, which implied that the JavaScript engine would *always* forcefully translate every JavaScript bytecode irrespective of whether it was hot or not. This resulted in a significant increase in performance as the benchmarks mostly involved hot functions in loops which benefited from removing the overheads of repeated interpretation attempts. On an average, the scores increased by 7.1x and the execution time reduced by 4.1x. The next experiment consisted of enabling both Interpretation and JägerMonkey (using `--no-ti --no-ion`) and this performed very similar to having JägerMonkey alone, mainly because the interpreter is used only as a fallback mechanism in case the translation fails.

With JavaScript supporting dynamic typing, it is important for the

runtime system to know the exact type of the object that it's working on. Otherwise, the JIT compiler needs to generate code that accounts for all the possible types of the involved values and significantly slows down the execution of the program. This is where the Type Inference engine comes into effect. This generates type information for the JavaScript program by monitoring the program code as well as tracking the types of values as the program executes. As long as the type of the object does not change, the translated code is still valid. In case the type changes, then the translated code has to be invalidated and the JavaScript engine has to re-interpret for the new object type. Having a good type inference engine is crucial to the performance of the SpiderMonkey JavaScript engine. The next experiment that was performed enabled the Interpreter, JägerMonkey and the Type Inference engine (using just the `--no-ion` flag). With the added benefit of Type Inference, the scores of all the benchmarks shot up considerably (an average of 2.2x across the entire suite) with the corresponding decrease in execution time. The outlier in this case was `RegExp` whose performance dropped by 14.2%. This can be attributed to the nature of the benchmark which looks for regular expression patterns over the 50 top websites. This benchmark is bound to have irregular behavior and highly variable types in which case the type inference analysis is just pure overhead and does not lead to forward progress in the benchmark.

The final experiment in this case was including IonMonkey as well. This is the current state of Firefox as it stands today, which uses the Interpreter, JägerMonkey, Type Inference and IonMonkey. As expected, the benefits of

having the optimizing compiler was immediately visible in the v8 benchmarks whose scores increased by an average of 23.8x compared to the interpreter alone and 1.35x compared to JägerMonkey and Type Inference. The execution times also decreased correspondingly showing the major impact that IonMonkey had in improving the performance of JavaScript programs in Firefox. Of particular interest is the drastic drop in performance of the benchmark `Splay` when IonMonkey was used. The benchmark consists of 3 main stages, viz. construction of the Splay tree, teardown of the Splay tree, and operations on the Splay tree. The construction phase is quite straightforward and creates n new nodes according to the size of the tree; the teardown phase makes a call to the garbage collector to clean up the unused nodes; the third phase involves doing a look-up and replacing the nodes with greatest key. This is not repetitive in nature, and the overhead of maintaining and translating the large splay tree object is very detrimental to the performance of IonMonkey which cannot reuse the generated code. The additional overhead of attempting to `IonCompile()` leads to the performance being even less than pure interpretation itself. This was an interesting special case while the rest of the benchmarks benefited spectacularly with the introduction of IonMonkey. This experiment also showed that more focus in required towards fine-tuning the performance and optimization passes implemented within IonMonkey in order to gain further benefits.

4.2.2 Time-sensitive Characterization of JavaScript programs

In order to thoroughly understand the behavior of JavaScript programs in different parts of the JavaScript engine, the source code was instrumented to collect the time trace of the entry and exit into the interpreter, both the JIT compilers and the garbage collector during the course of one execution. This fine-grained characterization is useful as it traces the flow of the JavaScript program. All the programs start off in the interpreter and after a certain period of time, control is transferred to the method-JIT (JägerMonkey) when a particular function becomes hot. The next transfer happens into the optimizing compiler (IonMonkey) when it is deemed that further optimization is required. In case the benchmark or website allocates too many objects and runs out of memory, the garbage collector is invoked to periodically clean up dead objects. This is the primary advantage of having the time-sensitive data, i.e. replicating the flow of the program through the different sections of the JavaScript engine. Figure 4.3 shows the execution plot for the benchmark `Crypto-AES.js` for a small time slice of execution across the different components (viz. Interpreter, JägerMonkey, IonMonkey and the Garbage Collector). The x-axis denotes the execution time (in milliseconds) and the y-axis fluctuates between a high and low state that indicates whether the particular component is being stressed or not.

This kind of time-sensitive information is also extremely valuable to study phase behavior in programs. This has been extensively studied by Sherwood et al. for SPEC benchmarks [28] but no such work has been done for

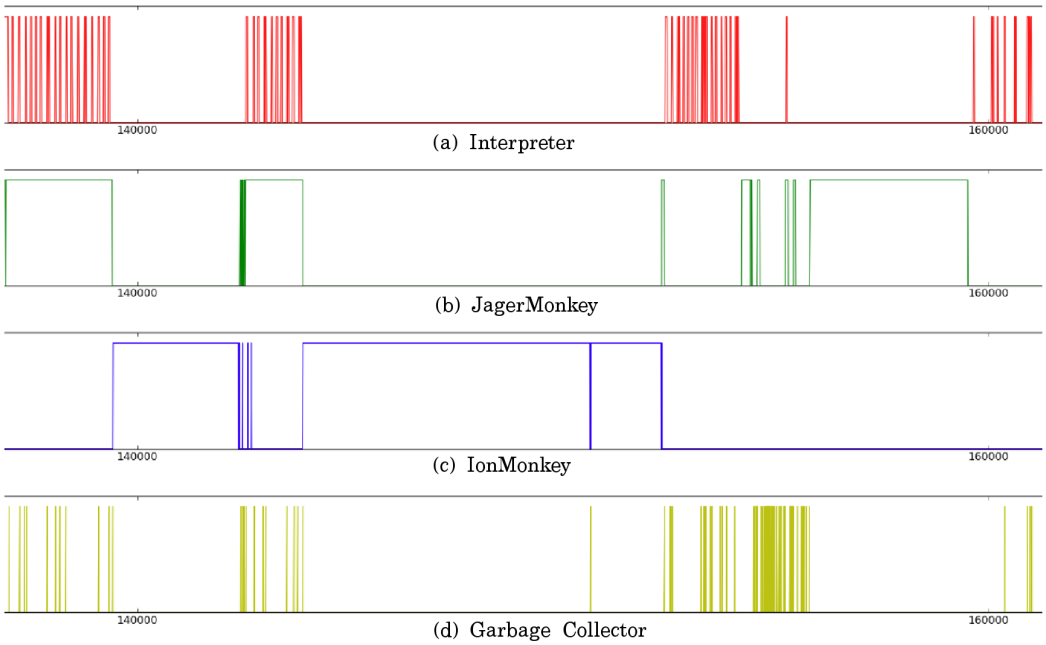

Figure 4.3: Plot showing activity in different parts of the JS Engine

JavaScript programs. However, phase analysis of JavaScript programs is beyond the scope of this thesis. In order to concisely show the entire execution of the JavaScript programs, the previous stack plot was modified and condensed into a single-color coded plot using the TraceLogger tool [12]. Each of the blocks denotes a time period of 2 milliseconds and the color of the block denotes which part of the JavaScript engine the control is currently in. The execution time increases from left to right. For the sake of brevity, only a few representative benchmarks from both the SunSpider and v8 benchmark suites are shown in Figures 4.4 to 4.9.

Figures 4.4 to 4.7 show a small subset of the Sunspider benchmarks

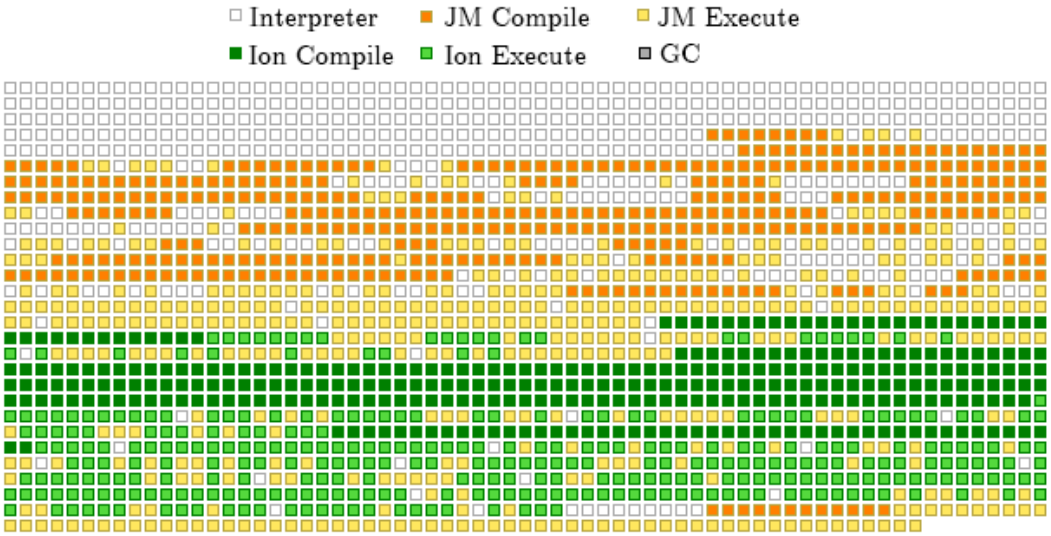

Figure 4.4: Component split-up: Sunspider - 3d-raytrace

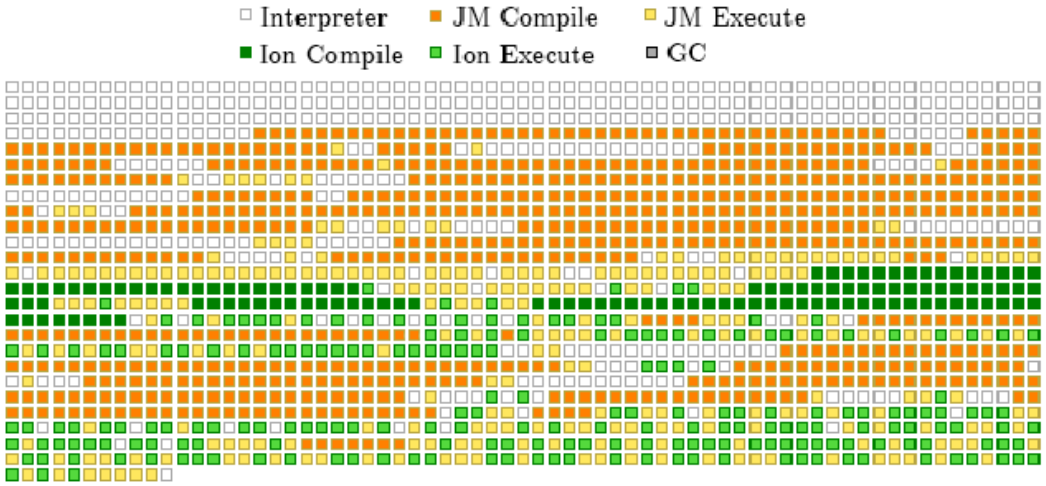

Figure 4.5: Component split-up: Sunspider - crypto-aes

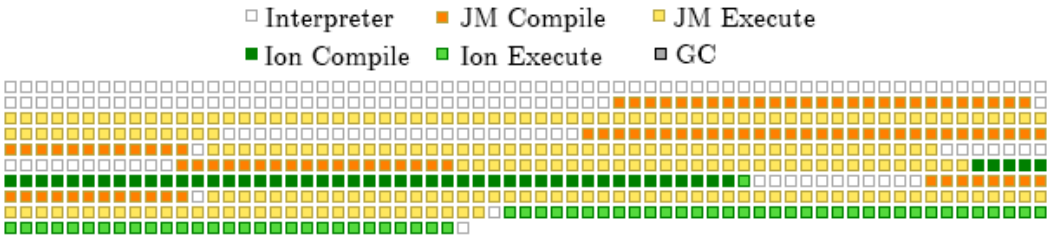

Figure 4.6: Component split-up: Sunspider - string-base64

Figure 4.7: Component split-up: Sunspider - math-cordic

as they trace their way through the SpiderMonkey engine. Initially, the time is spent in the Interpreter (shown as white boxes) before switching over to JägerMonkey (shown as orange/yellow boxes). Only a small portion of the time is spent in IonMonkey (shown as green/dark-green boxes) mainly because the Sunspider benchmarks are relatively short.

Figure 4.8 and Figure 4.9 show two v8 benchmarks as they trace their way through the SpiderMonkey engine. After a small stint in the Interpreter (shown as white boxes) and JägerMonkey (shown as orange/yellow boxes), a majority of the time is spent in IonMonkey (shown as green/dark-green boxes). In particular, it is observed that `Splay` also spends quite some time in the Garbage Collector which is in accordance to the explanation present in

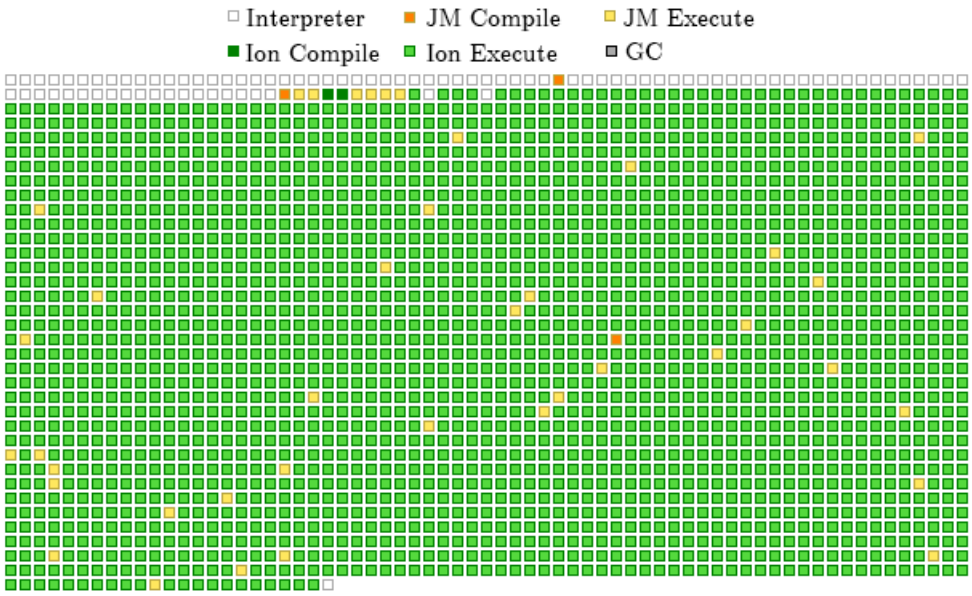

Figure 4.8: Component split-up: v8 - Richards

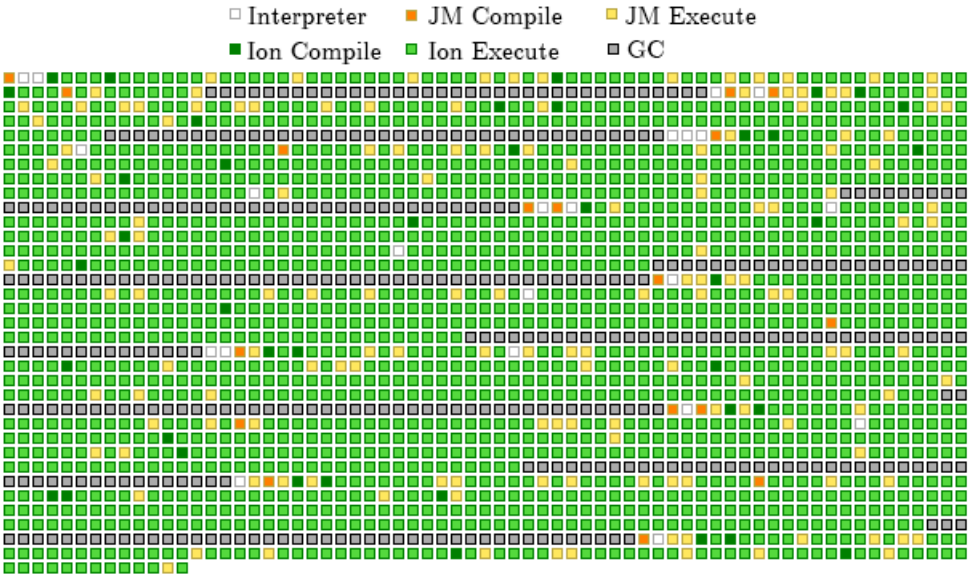

Figure 4.9: Component split-up: v8 - Splay

Section 4.2.1 purporting that a part of the benchmark is entirely devoted to cleaning up the Splay tree.

The next step involved studying the overall statistics across the entire execution. Figure 4.10 and Figure 4.11 show the time spent in each section of the JavaScript engine for the entirety of both SunSpider and v8 benchmark suites respectively. The detailed results are analysed in the following section.

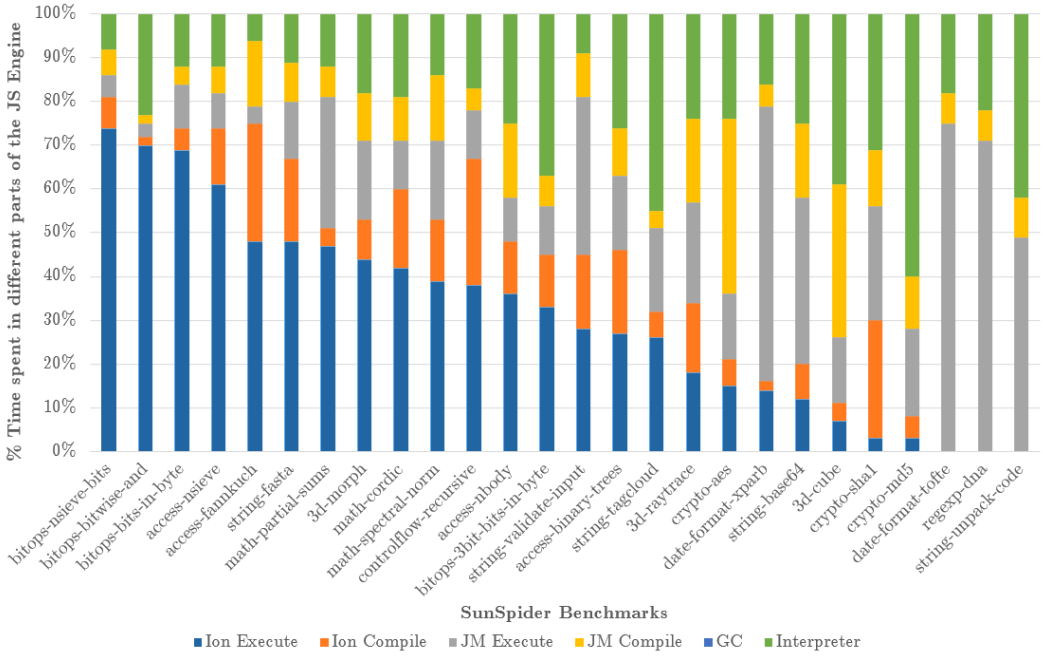

Figure 4.10: Component split-up summary: Sunspider Benchmarks

4.3 Results and Observations

From the results obtained, the following observations can be made. Sunspider benchmarks are relatively simple and have shorter execution times.

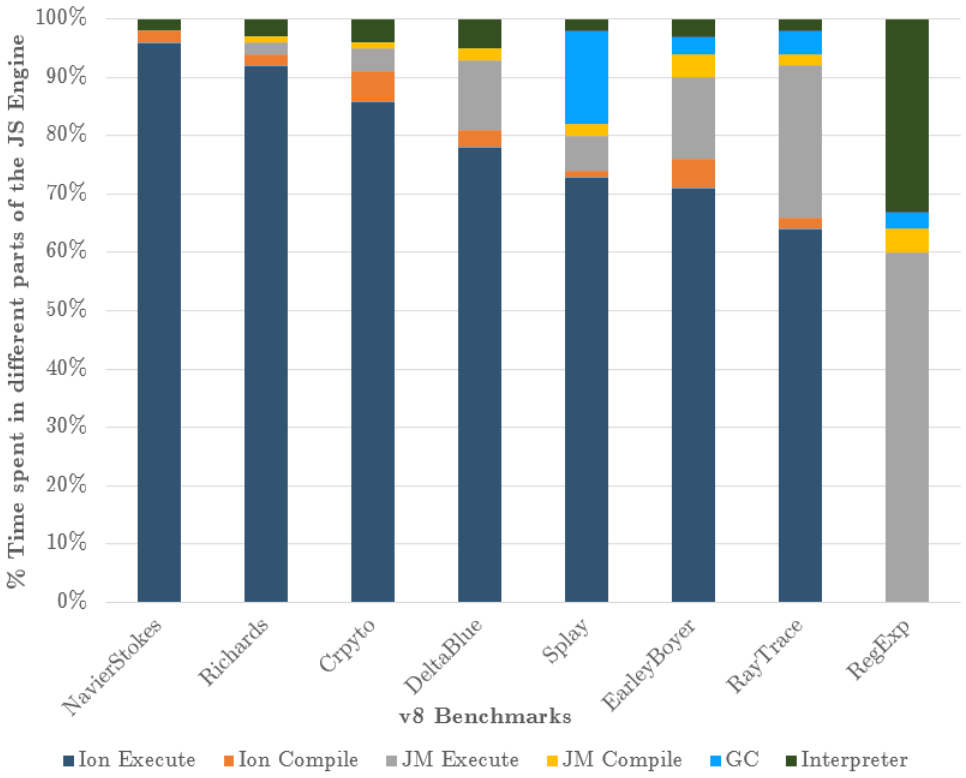

Figure 4.11: Component split-up summary: v8 Benchmarks

This leads to the fact that these benchmarks do not benefit greatly from the introduction of the IonMonkey optimizing compiler. By the time control is transferred to IonMonkey, the overheads of the optimization passes cannot be amortized because of the short execution times of these programs. This is supported by the fact that on an average, SunSpider benchmarks spend only 41.64% of their time inside IonMonkey while the majority of the time (58.36%) is spent in the Interpreter and JägerMonkey sections. Another interesting observation is that the garbage collector is never invoked (0%) during the middle of execution in the case of SunSpider benchmarks, which imply that

the benchmarks have a generally small working set size and do not create too many objects so as to necessitate a clean-up.

This is in stark contrast to the results obtained from the v8 benchmarks. These are relatively more complex and have longer execution times. Thus, the benefits of IonMonkey become more pronounced in these cases as the optimization overhead is amortized by the longer execution times of these programs. Across the entire v8 suite, a staggering 72.5% of the time is spent in IonMonkey, which is definitely a positive point considering that the optimized code is being run most of the times. A smaller fraction (17.5%) of the time is spent in JägerMonkey and a very small fraction (6.75%) is spent in the Interpreter. An interesting point to note in this case was that the code was being interpreted only in the very beginning, after which control transitioned into the IonMonkey. An exception to this trend was `RegExp` which spent 33% of time in the interpreter and 64% of time in JägerMonkey and did not even touch IonMonkey. This is because `RegExp` involves extracting the regular expressions from the 50 most popular websites, and does not have good code reuse. In fact, for regular expression benchmarks, the SpiderMonkey engine employs the services of the `YarrJIT` to aid in quicker processing. `YarrJIT` stands for "Yet Another Regex Runtime" which is a specialized regular expression processing engine developed by the Webkit team and was incorporated into the SpiderMonkey JavaScript engine. In addition, the v8 suite being longer running benchmarks, sometimes call the garbage collector as evidenced in `Splay`, `Earley-Boyer`, `RegExp` and `RayTrace`.

Using the time-sensitive trace of the benchmarks as well as the cumulative results, interesting observations were unearthed. Thus, by performing detailed characterization of these benchmarks with respect to the constituents of the JavaScript engine, a good scope for the opportunities for optimization is identified. This type of *compiler-centric* characterization of JavaScript programs is a major contribution of this thesis. This was also critical to the work carried out in the next chapter of the thesis, which focuses on a specific type of optimization to the Register Allocation heuristics within IonMonkey.

Chapter 5

Register Allocation

The previous chapter clearly showed the various avenues for optimization in the JavaScript engine; the fact that a majority of the time (72.5%) was spent in IonMonkey indicates that it is a great starting point for performing optimizations. In addition, as evidenced in Section 4.2.1, the performance improvements in SpiderMonkey were the greatest when IonMonkey was also included (as compared to using just JägerMonkey, Type Inference engine etc.). This highlights the importance of IonMonkey in this entire framework, and provides motivation to study this in further detail.

Another main takeaway was that longer running programs and benchmarks have a higher tolerance towards dynamic optimizations performed inside IonMonkey. Though some of the optimizations might have a larger overhead, it will be amortized over the long runtime of the program. Currently, IonMonkey implements the following optimization passes: Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), Dead Code Elimination (DCE), and Range Analysis. Since it is time critical, the register allocation scheme is Linear Scan Register Allocation (LSRA) as proposed by Poletto and Sarkar [26]. This chapter will focus on a specific optimization, viz. different

Register Allocation Heuristics for various types of JavaScript programs along with the associated tradeoffs in each case.

The main reason for picking Register Allocation was justified by performing a simple experiment. The time spent in the register allocation stage of IonMonkey was calculated and compared to the entire time spent in the backend optimizations (which includes translating to SSA, forming control flow graph and performing different optimization passes apart from register allocation). The results are presented in Table 5.1. We see that a majority of the time (52.81% on an average) is spent in the register allocation phase and the remaining time (47.19%) is spent in all the other stages combined. Thus, Register Allocation and its various heuristics were chosen as a poster child for this thesis.

Table 5.1: Time spent in Register Allocation stage in IonMonkey

Benchmark	**Reg Alloc (ms)**	**Overall (ms)**	**% Reg Alloc**
Crypto	63.201	110.809	57.03
DeltaBlue	42.331	100.072	42.30
EarleyBoyer	197.472	354.025	55.77
NavierStokes	68.390	165.069	41.43
RayTrace	34.668	78.337	44.25
RegExp	3.636	7.202	50.48
Richards	11.718	22.856	51.26
Splay	29.663	38.210	77.63

It is important to note that the ARM Cortex A9 in the Pandaboard is a dual issue out-of-order processor that has 16 general purpose registers R0

to R15, out of which some registers have specific purposes (R13 is the stack pointer, R14 is the link register and R15 is the program counter).

The remainder of this chapter is structured as follows: First, the necessary background and well-studied algorithms for register allocation is presented. Then, the implemented algorithm in SpiderMonkey is covered in considerable detail, along with supporting reasons. The next section focuses on the different register allocation heuristics that were implemented as part of this thesis, followed by the corresponding results and observations. The final section summarizes the main take-away of this study.

5.1 Background - Register Allocation

Register allocation is the process of assigning a large number of program variables to a small number of available CPU registers. There exist different granularities of register allocation - local register allocation within a basic block, global register allocation that extends over an entire function, and interprocedural register allocation that transcends function boundaries as well.

Local register allocation assumes that no registers are inherited from previous basic blocks and no values are left in the registers at the end of the basic block. The actual allocation can be done in a top-down or a bottom-up fashion. The top-down allocator works on the general idea that the most heavily used values are to be held in registers rather than going to the memory every single time to access them. On the first pass through the basic block, the usage statistics for all the virtual registers (virtual registers are infinite

in number and exist for each of the program variables in the generated code) are counted. Then, assuming that n physical registers are available on the system, the most used n virtual registers are assigned. For the remaining variables, appropriate load/store instructions are inserted on the second pass while maintaining the original program semantics. It is important to note that register assignment remains fixed for the entire lifetime of the basic block in consideration. The bottom-up allocator starts with an empty register set and loads variables into the registers on demand. When no register is available, one of the registers is spilled and the new value is assigned. The most common heuristic used for spilling is the one whos next use is farthest away in the future.

In order to obtain to find out when a particular variable will be used next, we need liveness information. Liveness analysis is a classic data flow analysis performed by compilers for each program point to calculate which variables may be potentially read before their next write, i.e. the variables are 'live' at the exit of each program point [1]. This is typically done as a backwards pass in reverse topological order of the control flow graph. In simple terms, a variable is live if it has a future use. From the point of view of bottom-up register allocation, a non-live value in a register can be discarded thereby freeing the register it had previously occupied. In most cases, the number of live variables is always more than the number of registers available and thus needs spilling. Spilling basically refers to the action of storing the value from a register to the memory in order to accommodate a new value.

The complementary action is called filling, where the value is restored from memory to the register.

The previous paragraphs were focused on local register allocation within a single block. But when we consider global register allocation across multiple blocks, it becomes more challenging especially because of the control flow complexity. To tackle this, multiple register allocation schemes have been proposed and two of the most popular ones are explained in detail below.

5.1.1 Register Allocation via Graph Coloring

Chaitin [4] proposed a classic algorithm for register allocation: he purported that the problem of register allocation is isomorphic to graph colorability. The problem of register allocation with k physical registers can be reduced to the problem of k-coloring a graph.

The graph in this case is called an interference graph, where the vertices are the set of all unique variables in the program, and the edges connect two vertices (variables) that are live at the same time. Some additional constraints may be applied on the graph, such as precolored vertices to denote general purpose registers vs. floating point registers. The algorithm works by assigning a color to vertex, then removing it from the interference graph until the graph is reduced to a single node or it cannot be colored further. At this stage, the values may need to be spilled and compensation code needs to be added. The implementation of this algorithm requires liveness analysis to be done to compute live ranges, requires the correct maintenance of the

interference graph, as well as a stack to push nodes that have been assigned a color already. It is an NP-complete problem, and is usually solved by applying certain heuristics based on the target platform.

However, in the case of dynamic systems like Just-in-Time compilers which are sensitive to high overheads and require fast register allocation, the implementation of this algorithm is not feasible. Most of the current compilers use Linear Scan which is discussed below.

5.1.2 Linear Scan Register Allocation

This algorithm was proposed by Poletto and Sarkar [26] and requires only a single pass over the list of live ranges of the variables. The general idea is that variables with short lifetimes are assigned to registers while those with longer lifetimes tend to be spilled. On an average, this algorithm only performs 12% less efficiently than the graph coloring algorithm but completes in linear time.

The steps involved for the Linear Scan Register allocator is as follows: First, a dataflow analysis is performed to gather liveness information and the live ranges are constructed and sorted in the order of increasing starting time. Then, the algorithm iterates through the live interval list and assigns a register from the available register pool. When the variables live range has ended, the register is freed and can be utilized for any of the upcoming live ranges. Since this algorithm runs in linear time, it is more suitable for dynamic environments like JITs. The remainder of this chapter focuses on the register allocation

mechanism implemented in the IonMonkey optimizing JIT and the various heuristics that were evaluated for this thesis.

5.2 Register Allocation in IonMonkey

This section first covers the motivation for looking closely into the register allocation scheme. Then the various register allocation heuristics and their details are explained. It is worthwhile to note that the register allocation discussed here in this section is particular to the IonMonkey backend optimizing JIT. The reason for focusing on IonMonkey is because long running benchmarks spend most of their time in IonMonkey as compared to the other parts of the SpiderMonkey JavaScript engine.

5.2.1 Motivation

IonMonkey's currently implements the Linear Scan Register Allocation (LSRA) scheme. It is unclear whether this algorithm is optimized within IonMonkey because of the fact that the entire system is still in its infancy. Also, there haven't been previous studies that show linear scan is the optimal register allocation scheme for JavaScript applications apart from the known fact that LSRA is preferred in JIT compilation over graph coloring. This thesis focuses on exploring both of these questions.

The ultimate goal is to gain knowledge of how JavaScript applications interact with the JavaScript engine, and which register allocation heuristics suit them the best. To determine the optimal algorithm, a wide range of

register allocation algorithms were implemented that vary in complexity, code size, and optimization level, capturing a wide spectrum of heuristics. By using such a set of incremental algorithms, the algorithm with the optimal trade-off between these heuristics by creating the optimal compile time and runtime can be identified.

First, the traditional graph coloring register allocation algorithm was implemented within the IonMonkey framework. In addition, various heuristics of the existing LSRA were changed. By measuring the relative performance of each implementation on a representative set of JavaScript benchmarks, potential areas for possible improvement within the existing LSRA can be located. The different heuristics are classified into three different categories: the first focuses on various techniques for register assignment, i.e. choosing which register to assign to the variable; the second category focuses on which live interval to spill in case there are no more free registers; the final category is the interval priority heuristics which covers different possible orders in servicing live intervals before the actual assignment takes place. Each of these categories are explicated below.

5.2.2 Register Assignment Heuristics

Register assignment is the step within register allocation that determines the appropriate physical register to assign the given live interval. As mentioned in the previous sections, the original IonMonkey uses the Linear Scan Register Allocation (LSRA) for this step of register allocation. In LSRA,

the physical register with the furthest away next interval is chosen for the interval assignment. For this study, numerous alternative register assignment protocols were implemented. For a complete list of the register assignment algorithms implemented and a brief description of each, refer to Table 5.2 below.

Table 5.2: Register Assignment Heuristics

Algorithm	**Description**
Linear Scan (Default)	Furthest next interval
Graph Coloring	Interval interferences in dependence graph
Trivial	First available register at interval's start
Number of Ranges	Fewest number of intervals
Range Size	Smallest overall range size
Subset of Registers	Linear scan with limited registers

First, the graph coloring register assignment algorithm was implemented. In graph coloring, the algorithm first constructs a dependency graph, where each node represents a live interval and the edges represent interferences between two live intervals (i.e. the overlap of the intervals' live ranges). Then, the algorithm selects live intervals, one at a time, which have less interferences than the number of physical registers available, and removes the from the graph, pushing them onto a stack. This process continues until no such live interval exists, in which case a register is chosen to be spilled based on certain spill-cost heuristics. Once every node has been removed, the algorithm pops the live intervals from the stack one at a time, reinserting them into the original graph and assigning all the 'non-spilled' nodes a color such that no

two interfering nodes have the same color. This color represents a physical register assignment, thus spilled nodes are not assigned a color.

In the implementation of graph coloring register assignment, the live intervals were serviced using the same priority as linear scan, in order of earliest start time. This heuristic was maintained in order to compare the effects of the graph coloring and linear scan algorithms themselves, not their underlying priority heuristic. The heuristic used for determining an interval to spill was to choose the interval in the current graph with the most interferences.

It was found that the graph coloring implementation is much more complex than LSRA in terms of time as well as having a substantially larger in code size, both static and dynamic, due to the additional data structures.

The *trivial*, *Number of Ranges*, and *Size of Ranges* register assignment schemes are less complex implementations than linear scan and graph coloring. 'Trivial' assignment is the least complex and thus smallest in space and fastest in compilation time. This assignment algorithm simply chooses the first free register (sequentially) and assigns it to the current interval. 'Number of Ranges' assignment selects the physical register with the least number of ranges currently assigned to it, provided this register is available at the start of the interval to be assigned. Similarly, 'Size of Ranges' assignment selects the physical register with the smallest overall interval currently assigned to it, given the same provisions. These two assignment heuristics are meant to represent algorithms between those of trivial assignment, Linear Scan and Graph Coloring in terms of complexity, code size, and compile time.

The 'Subset of Registers' assignment algorithm is identical to LSRA except that it restricts the compiler to a subset of the overall register count. For the purpose of this thesis, experiments were conducted by setting the number of registers to 4, 8 and the default value of 16. Intuitively, this algorithm should not perform better than the original LSRA algorithm. By measuring the impact on the overall performance results for the various benchmarks, the sensitivity of JavaScript programs to this register restriction can be studied. This can provide vital information about JavaScript applications and their register use patterns.

5.2.3 Interval Spill Heuristics

Interval spill heuristics refer to the decision of which interval to spill to memory when no registers are available for an interval's assignment. The original IonMonkey implementation spills an interval based on the next use of each interval (referred to as 'SpillNormal'). Many other heuristics were implemented that vary in complexity, code size, and runtime. A trivial implementation (in terms of these characteristics) as to always spill the interval that is already assigned (referred to as 'SpillBlocked'). Two more complex implementations involved spilling the interval with lesser uses (referred to as 'SpillNumUses') and spilling the interval with the larger range size (referred to as 'SpillRangeSize'). The summary of these heuristics is present in Table 5.3. Again, these heuristics were chosen in an attempt to capture a wide range of complexity in order to tradeoff the benefits from such optimizations for

JavaScript applications in terms of performance, code size, and speed.

Table 5.3: Interval Spill Heuristics

Algorithm	**Description**
Spill Normal (Default)	Based on next use
Spill Blocked	Spill interval that is already assigned
Spill Num Uses	Spill interval with least uses
Spill Range Size	Spill interval with largest range size

5.2.4 Interval Priority Heuristics

Interval priority heuristics refer to the ordering in which intervals are serviced by the register assignment step explained in Section 5.2.2. For this phase of register allocation, the study aims to characterize JavaScript applications sensitivities as well as IonMonkeys LSRA sensitivities. In the original IonMonkey LSRA, intervals are serviced in order of start position, where intervals with earlier start positions are assigned first. Numerous other heuristics were implemented for this step in the register allocation process - one that services intervals in order of latest end time (referred to as LinearScanEndTime), one in order of number of intervals where virtual registers with more live intervals are serviced first (referred to as NumRanges), and one in order of live interval size (referred to as RangeSize). The first heuristic variation aims to capture register usage patterns while the second and third focuses on the temporal locality of JavaScript applications. The summary of these heuristics are present in Table 5.4.

Table 5.4: Interval Priority Heuristics

Algorithm	**Description**
Start Time (Default)	Earliest start times are serviced first
End Time	Latest end times are serviced first
Number of Ranges	Variable with more live ranges are serviced first
Range Size	Variable with larger live ranges are serviced first

5.3 Results and Observations

Before detailed analysis of the results, it is worthwhile to mention another tool that was used to validate the correctness of the implemented heuristics. c1Visualizer [20] is a GUI tool written in Java that processes the debug information spewed out by IonMonkey and displays valuable information such as control flow graphs, intermediate code, live interval ranges etc. For this thesis, the Live Interval graphing tool was leveraged to verify the different register assignments for the different heuristics that were implemented.

Figures 5.1 and 5.2 show two v8 benchmarks along with their register assignments as displayed by the c1Visualizer tool (the rest of the figures for other benchmarks have been omitted for brevity). The y-axis represents all possible variables in the program and the x-axis represents the execution time and also has additional information as to which basic block is being executed.

Figure 5.1 shows the register allocation for a particular run of `DeltaBlue` using the *Trivial* scheme. In the beginning, it can be observed that register `rax` is assigned to the first live interval, followed by `rcx` for the next live inter-

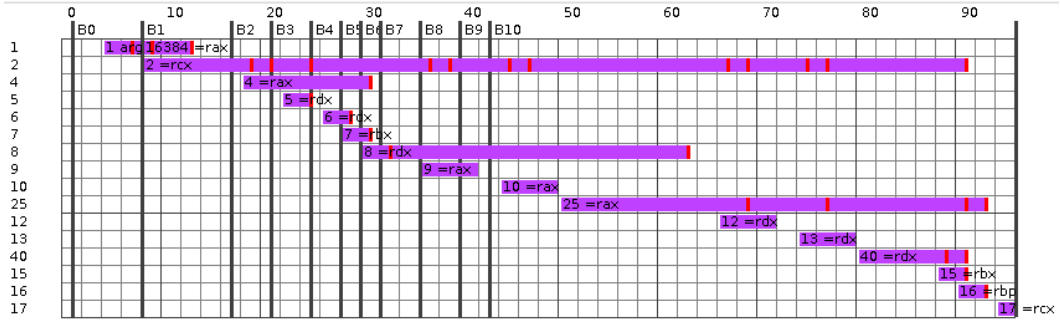

Figure 5.1: Live Ranges in DeltaBlue using c1Visualizer

val. Subsequently, when the next interval is processed, `rax` is assigned again as it is the first register available in the free register list. One can also easily observe that `rax` is the most commonly assigned register, which occurs for a series of live intervals later in the execution. The fact that `rax` is always picked first instead of iterating through the free register list is the main idea behind the *trivial* allocation scheme.

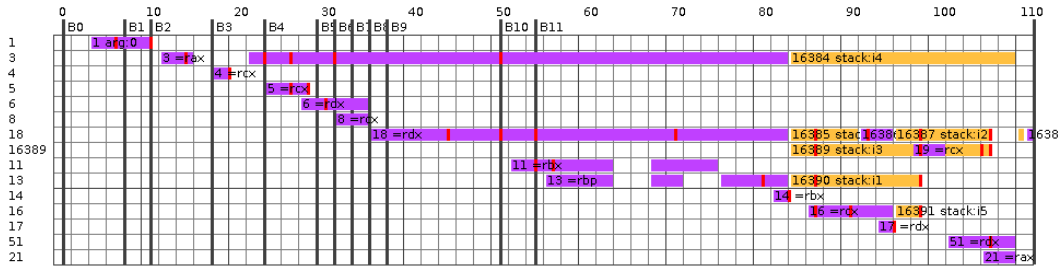

Figure 5.2: Live Ranges in EarleyBoyer using c1Visualizer

Figure 5.2 shows the register allocation for a particular run of `Earley Boyer` using the *SpillRangeSize* scheme. This heuristic, as explained in the previous section, chooses to spill variables with the largest live range size, as it tends to block a particular register if assigned. In this case, live ranges *3*

and *16389* have the largest sizes, and are thus spilled to the stack (denoted by the yellow shade). In conclusion, the c1Visualizer helped ensure that the implemented heuristics were working as expected.

The following subsections focus on the results obtained for each category of heuristics.

5.3.1 Results - Register Assignment Heuristics

The v8 benchmark scores obtained from the four register allocations schemes can be seen in Figure 5.3. The Trivial scheme performs, on an average, 11.89% better than the default IonMonkey Linear Scan Register Allocation (LSRA) scheme. The highest performance increase was seen in Navier-Stokes, with a solid 27.72% improvement, followed by Richards at 21.51%, and Crypto at 20.86%. This is mainly because these three benchmarks spend a lot more time in the register allocation stage as shown in Table 5.1 and thus benefit the most from the simpler allocation scheme. RayTrace performed the worst with a 4.27% degradation when compared to default LSRA. The unique degradation of RayTrace implies an inherent heavy and/or complex register usage pattern compared to the other benchmarks in the suite. The large improvement in performance for many of the benchmarks suggests that complex register allocation schemes are not necessary for many real-world JavaScript programs and can actually degrade performance due to the large computational overheads. Often, a simpler technique such as trivial assignment will suffice. To determine this tradeoff at compile time, a smart decision making

system could be instated to choose between trivial allocation and LSRA based on the workload characteristics.

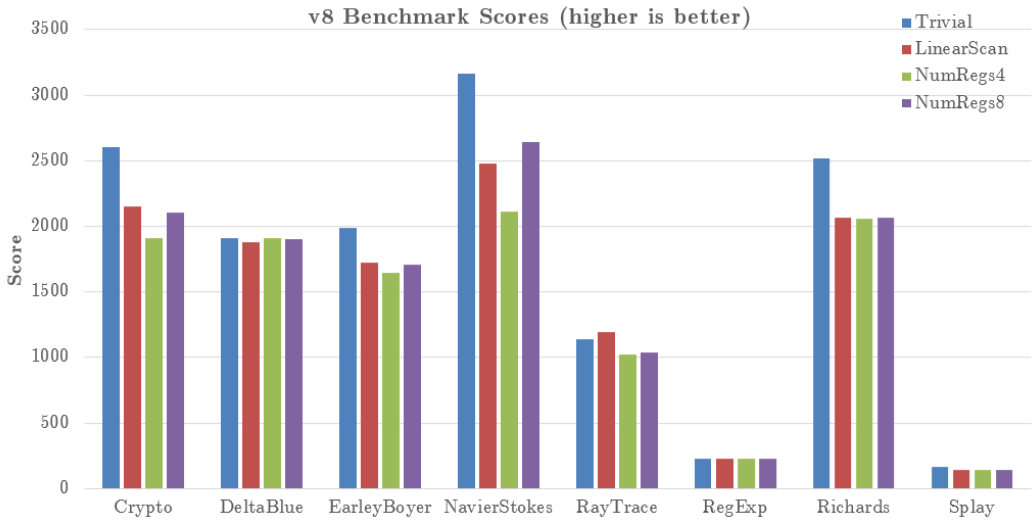

Figure 5.3: Results - Register Assignment Heuristics

This data also provided key insight into JavaScript benchmarks and the applications they represent. The register allocation scheme that limits the number of registers from the full 16 to 4 and 8 perform extremely poorly in comparison to LSRA with an average degradation of 6.65% and 1.9% respectively. These results suggest that a lesser number of available registers will hinder register allocation significantly. The more significant performance hit is incurred when limiting the registers from 8 to 4 than from 16 to 8. This finding further implies that most v8 JavaScript benchmarks, and thus the applications that they are built to represent, are highly register intensive and take advantage of the full register count of 16. It is also important to note the large difference in performance scores between RegExp, a low scoring

benchmark, and NavierStrokes, one of the highest. Although both of these benchmarks are based on array and string manipuation, their scores are significantly different. This difference suggests that JavaScript performance is highly sensitive to array and string manipulations in workloads.

5.3.2 Results - Interval Spill Heuristics

The results for the different interval spill cost heuristics applied to the standard LSRA are shown in Figure 5.4. The heuristic based on largest interval size shows a 13.52% overall increase in performance. Crypto shows an impressive performance improvement of 68.72% with this heuristic. The heuristic on number of uses also shows a significant performance increase, with an average of about 18.13%. The benchmarks most benefited by the SpillNumUses heuristic are Crypto(84.01%) followed by NavierStokes(38.33%).

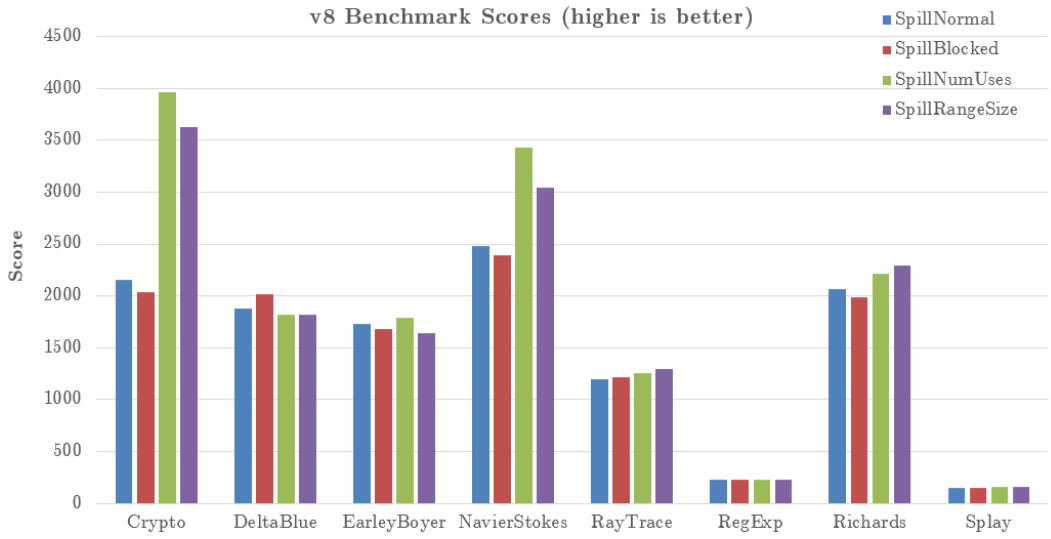

Figure 5.4: Results - Interval Spill Heuristics

These results suggest that JavaScript applications have a high number of large live intervals that do not contain a lot of uses. This makes these intervals top candidates to be spilled because they free up a large interval of a physical register and only incur minimal overhead for subsequent uses. Thus, when these heuristics are used over IonMonkey's 'Next Use' heuristic, we see drastic performance improvements. It can be seen that Crypto and NavierStokes leverage the SpillNumUses and SpillRangeSize heuristics most effectively. These benchmarks each stress different JavaScript features as described in the section 3.3, suggesting that a wide spectrum of JavaScript applications would benefit from these types of spill heuristics over IonMonkey's current implementation. The heuristic of always spilling the blocked register is at par with the default IonMonkey spill heuristic. Similar to the findings in the Register Allocation Schemes in Section 5.2.2, this suggests that the linear scan algorithm may not produce enough performance benefits over a trivial assignment to warrant its time and space complexity.

5.3.3 Results - Interval Priority Heuristics

Plots of the priority heuristics are shown in Figure 5.5. Overall, the Linear Scan register allocation based on interval end times peforms 1.2% worse than the default LSRA. This suggests that the live interval characteristics at the startup and startdown phases of JavaScript benchmarks do not have any distinct differences between them. The priority heuristic 'Number of Ranges' shows a 6.12% improvement. The heuristic 'Range Size' performs the worst

with a 6.25% degradation in performance overall. This further supports the spill cost heuristic findings that JavaScript applications tend to have large live intervals with a relatively few number of uses.

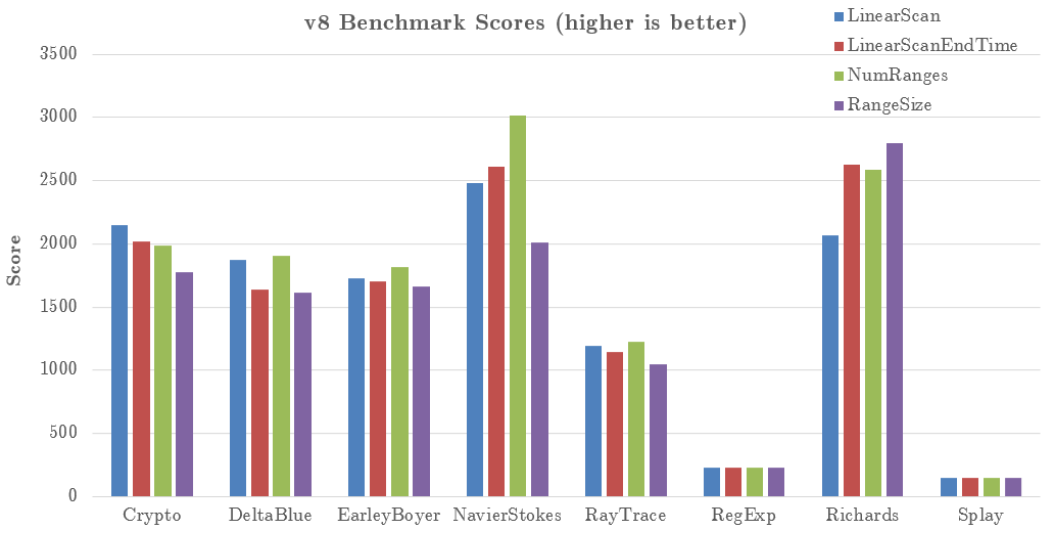

Figure 5.5: Results - Interval Priority Heuristic

This results in a performance degradation because registers are dedicated to intervals which are idle, while other smaller live intervals are not able to be allocated and thus spilled, incurring time costs. Overall, the results show that JavaScript performance is not sensitive to the order in which the live intervals are processed, or assigned to physical registers. IonMonkey's current priority heuristic appears as one of the optimal solutions for the examined benchmarks.

Thus, by carefully studying different register allocation heuristics in a systematic and incremental fashion, this thesis has contributed towards gath-

ering information about JavaScript programs with respect to their register allocation needs.

5.4 Towards an "Optimal" Register Allocation Scheme

After conducting the set of experiments described in the previous section, the best heuristics were combined statically to create an "optimal" register allocation scheme. Ideally, a dynamically adaptive scheme that changes between different heuristics depending on the characteristics of the application would work the best. However, for the purposes of this thesis the "best" heuristics were statically selected to observe the maximum possible benefit assuming no overhead in switching between different heuristics.

As shown in the previous section, the *Trivial* register allocation scheme worked the best amongst the different register allocation schemes and the *SpillNumUses* spill heuristic worked best amongst the different spill heuristics. This last experiment compared the performance of the default linear scan against the combined optimized heuristics - *Trivial+SpillNumUses*.

Figure 5.6 shows the performance numbers of both the techniques. We can see that the optimal heuristic performs 9.1% better than the default Linear Scan algorithm on an average across the entire v8 benchmark suite. Some of the benchmarks like `RegExp` and `Splay` show no improvement as compared to the default Linear Scan while `RayTrace` performs 3.2% worse. This can be mainly attributed to the inherent nature of RayTrace which has a complex register-usage to correctly render the ray tracing scene. The noticeable im-

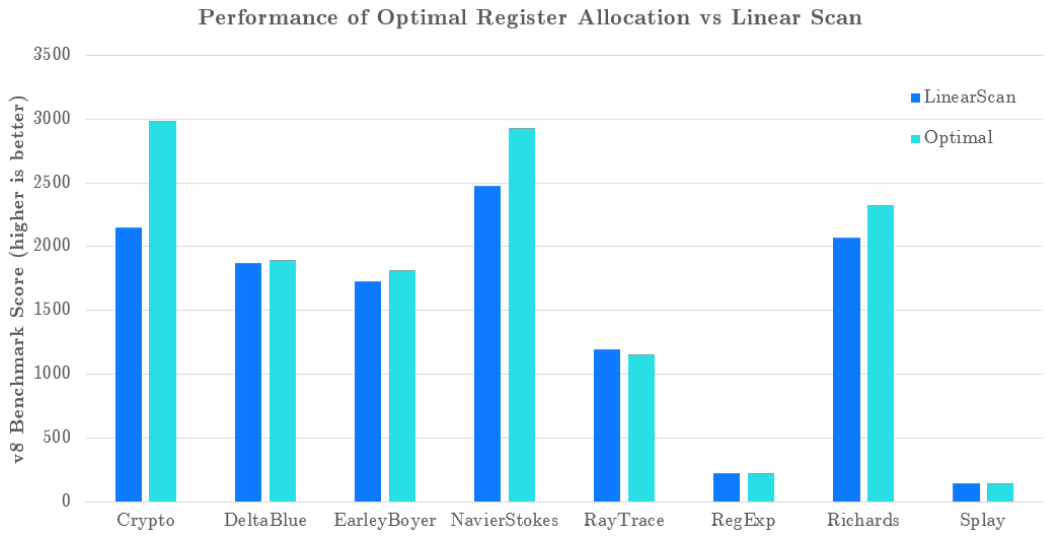

Figure 5.6: Results - Optimal Register Allocation vs Linear Scan

provements are by `Crypto` and `NavierStokes` which show a 36.6% and 18.1% increase in performance respectively. These two benchmarks were the ones that benefited most from the *SpillNumUses* heuristic, and thus show an improvement in the optimal register allocation scheme as well.

Table 5.5 shows the execution time of these benchmarks for both the tested schemes in milliseconds. On an average, the optimal register allocation scheme finished execution 11.23% faster than the default scheme. However, it is clear that the main improvement is for benchmarks like Crypto, EarleyBoyer and NavierStokes in which the execution time difference is significant enough to be noticed. Since these benchmarks are modeled to reflect real pages, any webpages that have characteristics inherent of these benchmarks is bound to show an improvement in performance too.

Table 5.5: Execution Time Comparison - Optimal Scheme vs Linear Scan

Benchmark	**Linear Scan (ms)**	**Optimal (ms)**
Crypto	5792	4673
DeltaBlue	2040	2018
EarleyBoyer	7990	7160
NavierStokes	4490	2940
RayTrace	1655	1712
RegExp	15509	15494
Richards	2505	2326
Splay	7278	7186

In order to verify the effectiveness of the "optimal" allocation scheme, the time spent in register allocation versus the overall time in the IonMonkey backend compiler was verified again in an experiment similar to Table 5.1. The results of this experiment are presented in Table 5.6.

Table 5.6: Time spent in Register Allocation stage - Optimized version

Benchmark	**Reg Alloc (ms)**	**Overall (ms)**	**% Reg Alloc**
Crypto	57.971	106.754	54.30
DeltaBlue	30.150	72.052	41.84
EarleyBoyer	154.198	277.769	55.51
NavierStokes	42.339	109.192	38.77
RayTrace	57.679	90.517	63.72
RegExp	3.586	7.225	49.42
Richards	5.481	15.717	34.87
Splay	28.716	43.761	65.62

The first observation is that the overall runtime of IonMonkey has been

reduced from the default implementation of Linear Scan. Over the entire set of v8 benchmarks, IonMonkey with the default Linear Scan scheme took 109.56ms while IonMonkey with the optimized scheme took just 90.36ms. This can be primarily attributed to the optimal register allocation heuristics because the rest of the optimizations passes were the same as the original.

In addition, the percentage of time spent in the Register allocation stage has also reduced. On an average across the entire v8 benchmark suite, the register allocation stage is only 50.51% of the IonMonkey backend (as compared to 52.81% in the default Linear Scan implementation, *lesser is better in these cases*). Some of the better improvements are observed in `Richards` (51.26% in default versus 34.87% in optimized), `Crypto` (57.03% in default versus 54.30% in optimized) and `NavierStokes` (41.43% in default versus 38.77% in optimized). This clearly shows that lesser time is spent in the register allocation stage thereby leading to overall faster execution time.

Thus, by performing the optimal register assignment heuristic and evaluating it against the default Linear Scan implementation, a potential improvement in performance is identified. Ideally, if a dynamically adaptive system was in place, it could leverage the various heuristics much better than the statically selected "optimal" heuristic. This will definitely aid future work in tuning the IonMonkey optimization passes in order to extract maximum performance from the SpiderMonkey engine.

Chapter 6

Summary and Conclusions

JavaScript is at the epicentre of Web development today and has a massive outreach and impact. It is becoming increasingly important to optimize JavaScript performance on browsers for both current generation websites as well as the future HTML5 based websites. This is all the more crucial in the context of mobile systems which have limited processing capabilities and are energy limited. This thesis presented a detailed characterization work of JavaScript programs followed by register allocation optimizations on a representative mobile system.

Existing work on characterizing JavaScript such as [27] and [31] no not focus on the JavaScript engine itself. They are mainly focused on the properties of the JavaScript program and the impact of these programs on microarchitecture dependent statistics (such as IPC, branch misprediction rate, cache miss rate etc.) as well as microarchitecture independent statistics (such as ILP, control-flow predictability, instruction/data locality etc.). This thesis presents a *compiler-centric* characterization of the JavaScript programs as it interacts with the JavaScript engine. The results are exploited to make further optimizations in IonMonkey where it is found that a majority (72.5%) of

the time is spent. The detailed characterization also presents other insights such as shorter programs do not effectively utilize the IonMonkey optimizing JIT as the overheads are too high to be amortized. In addition, most of the benchmarks do not stress the garbage collector (0% in SunSpider and an average of 3.25% in v8). Using the TraceLogger tool, the time-sensitive trace of JavaScript programs was also identified. The general trend of these programs was to start off in interpretation-mode and then switch to one of the JITs (either JägerMonkey or IonMonkey) for faster execution. There were some exceptions to this case (`RegExp`) which could not benefit at all from IonMonkey due to the complexity in finding regular expressions which generates very little code-reuse. However, on an average, the IonMonkey compiler was active for 72.5% of the time spent in the v8 benchmark suite.

Leveraging the results of the characterization work, a specific optimization was applied on IonMonkey in which different register allocation heuristics were tried. In this thesis, there were three avenues of focus: *register assignment heuristics*, i.e. choosing which register to assign to the variable; *interval spill heuristics*, to identify which interval to spill in case there are no more free registers; and *interval priority heuristics* which covers different possible orders in servicing live intervals before the actual assignment takes place. By systematically evaluating these different heuristics and comparing with the existing baseline of Linear Scan Register Allocation (LSRA), interesting results were observed. The *trivial* allocation scheme, which picks the first available free register, yielded a large improvement in performance compared to the base-

line LSRA for many of the benchmarks. This suggested that complex register allocation schemes are not necessary for many real-world JavaScript programs and oftentimes simpler techniques are beneficial. Another experiment which varied the number of physical registers available for allocation (from 16 to 8 to 4) also yielded stimulating results. The performance drop from 8 registers to 4 registers was much higher than the corresponding drop from 16 registers to 8 registers. This showed that most of the v8 JavaScript benchmarks are highly register intensive and are very sensitive to a drop in register count.

In order to test the benefit of these heuristics, the best choice of heuristics (viz. *Trivial* and *SpillNumUses*) was statically selected and pitted against the default Linear Scan implementation. It was observed that there was a performance increase of 9.1% and execution time decrease of 11.23% across the entire v8 suite of the optimal scheme over the default implementation. This also proves a point that having a dynamically adaptive scheme which chooses different heuristics depending on the characteristics of each benchmark would definitely be a huge benefit within IonMonkey.

This also sets the stage for future optimizations to IonMonkey, and the entire SpiderMonkey engine as a whole. This thesis mainly focused on Register Allocation, and there is a lot of scope available in fine-tuning the other optimizations as well. Though the implementation was done with the JavaScript engine in Firefox, the concepts can be carried across to other implementations as well. The characterization work is platform-agnostic and the results can be leveraged in optimizing other JavaScript engines too.

In summary, this work concludes that by systematically characterizing JavaScript programs and performing corresponding optimizations, a compelling improvement can be seen in existing implementations. However, using these principles in developing a novel JavaScript engine from ground-up presents a significant challenge to compiler designers and language experts to come up with both high-performance and energy-efficient systems.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] asm.js. Latest specification of asm.js. `http://asmjs.org/spec/latest/`, 2013.

[3] BuiltWith. Javascript usage statistics. `http://trends.builtwith.com/javascript`, June 2010.

[4] Gregory Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Not.*, 39(4):66–74, April 2004.

[5] Netscape Communications. Netscape navigator. `http://en.wikipedia.org/wiki/Netscape_Navigator`, 1995.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical report, Providence, RI, USA, 1991.

[7] eBiz MBA. Top 10 best html5 websites of 2013. `http://www.ebizmba.com/articles/best-html5-websites`, 2013.

[8] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[9] Mozilla Foundation. Spidermonkey javascript engine. `https://developer.mozilla.org/en-US/docs/SpiderMonkey`.

[10] Mozilla Foundation. Rhino javascript engine. `https://developer.mozilla.org/en-US/docs/Rhino`, 2006.

[11] Mozilla Foundation. Ionmonkey in firefox 18. `https://blog.mozilla.org/javascript/2012/09/12/ionmonkey-in-firefox-18/`, 2012.

[12] Mozilla Foundation. Trace logger. `http://www.alasal.be/ionmonkey/`, 2012.

[13] Mozilla Foundation. Mozilla is unlocking the power of the web as a platform for gaming. `https://blog.mozilla.org/blog/2013/03/27/mozilla-is-unlocking-the-power-of-the-web-as-a-platform-for-gaming/`, 2013.

[14] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings*

of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[15] Philip Greenspun. Netscape livewire. `http://philip.greenspun.com/wtr/livewire.html`, 1998.

[16] Meteor Development Group. Meteor js. `http://meteor.com/`, 2012.

[17] Google Inc. V8 javascript engine. `https://code.google.com/p/v8/`, 2008.

[18] Google Inc. V8 benchmark suite. `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`, 2011.

[19] Joyent Inc. Node.js. `http://nodejs.org/`, 2009.

[20] Java.net. c1 visualizer for ionmonkey. `http://java.net/projects/c1visualizer/`, 2012.

[21] David Mandelin. Jägermonkey: the "halfway" point. `https://blog.mozilla.org/dmandelin/2010/05/10/jm-halfway/`, 2010.

[22] M. Mehrara, Po-Chun Hsu, M. Samadi, and S. Mahlke. Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 87–98, 2011.

[23] Nicholas Nethercote. Generational garbage collectors. https://blog.mozilla.org/nnethercote/category/garbage-collection/.

[24] Pandaboard. Pandaboard development platform. http://pandaboard.org/.

[25] Pandaboard. Pandaboard es specifications. http://pandaboard.org/content/pandaboard-es, 2011.

[26] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[27] Paruj Ratanaworabhan, Benjamin Livshits, Benjamin Zorn, and David Simmons. Jsmeter: Measuring javascript behavior in the wild, 2010.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM.

[29] Matt Silverman. The history of html5. http://mashable.com/2012/07/17/history-html5/, 2012.

[30] Statista. Statistics and facts about smartphones. http://www.statista.com/topics/840/smartphones/, 2012.

[31] D. Tiwari and D. Solihin. Architectural characterization and similarity analysis of sunspider and google's v8 javascript benchmarks. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 221–232, 2012.

[32] W3C. Document object model (dom). `http://www.w3.org/DOM/`, 2005.

[33] Webkit. Squirrelfish javascript engine. `http://trac.webkit.org/wiki/SquirrelFish`, 2005.

[34] Webkit. Sunspider benchmark suite. `http://www.webkit.org/perf/sunspider/sunspider.html`, 2007.